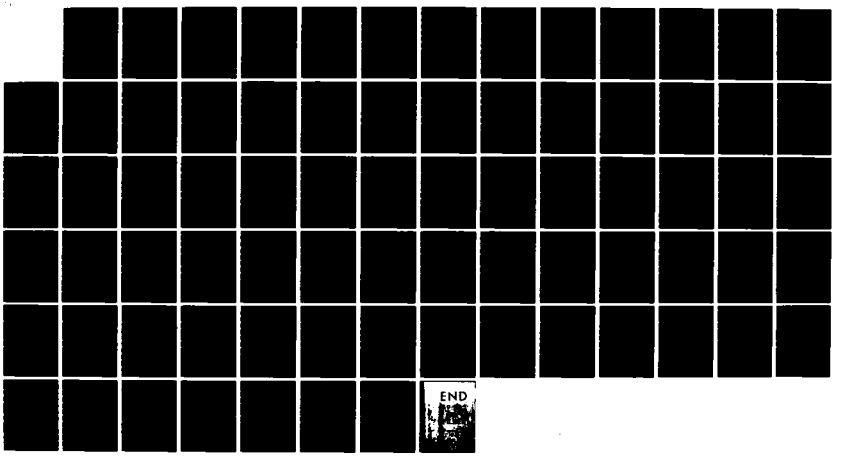
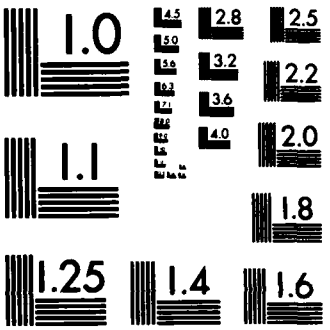
AD-A132 611 LEARNING TO PROGRAM IN LISP(U) CARNEGIE-MELLON UNIV 1/1
PITTSBURGH PA DEPT OF PSYCHOLOGY J R ANDERSON ET AL.
01 SEP 83 TR-83-1-ONR N00014-81-C-0335

UNCLASSIFIED F/G 9/2 NL





Learning to Program in LISP[1]

John R. Anderson
Robert Farrell
Ron Sauers
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

This research was supported by the Personnel and Training Research Programs,
Psychological Services Division, Office of Naval Research, under Contract
No.: N00014-81-C-0335, Contract Authority Identification Number,
NR No.: 157-465 to John Anderson.

Unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER TR ONR 83-1	2. GOVT ACCESSION NO. AD-A132611	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Learning to Program in LISP		5. TYPE OF REPORT & PERIOD COVERED Final Technical Report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) John R. Anderson Robert Farrell Ron Sauers		8. CONTRACT OR GRANT NUMBER(s) N00014-81-C-0335
9. PERFORMING ORGANIZATION NAME AND ADDRESS Department of Psychology Carnegie-Mellon University Pittsburgh, PA 15213		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS NR 157-465
11. CONTROLLING OFFICE NAME AND ADDRESS Personnel and Training Research Programs Office of Naval Research Arlington, VA 22217		12. REPORT DATE Sept. 1, 1983 13. NUMBER OF PAGES 52
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

production systems	computer simulation	knowledge compilation	retrieval
programming	analogy	proceduralization	planning
LISP	working memory	composition	
problem-solving	cognitive skill	problem decomposition	
goal structures	skill acquisition	automatic programming	

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We have gathered protocols of subjects in their first 30 hours of learning LISP. The process by which subjects write LISP functions to meet problem specifications has been modeled in a simulation program called GRAPES (Goal Restricted Production System). The GRAPES system embodies the goal-restricted architecture for production systems as specified in the ACT* theory (Anderson, 1983). We compare our simulation to human protocols on a number of problems. GRAPES simulates the top-down, depth-first flow of control exhibited by subjects

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

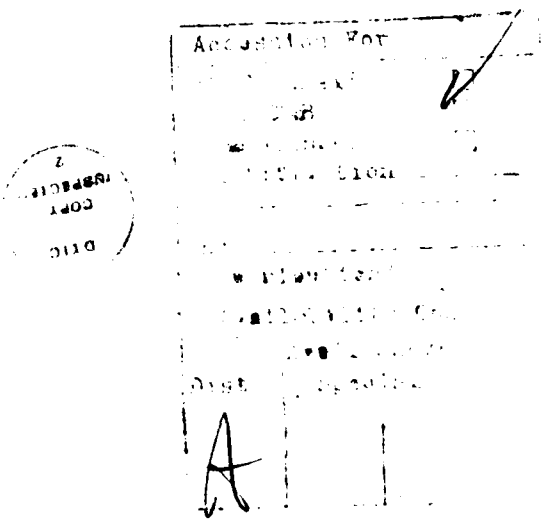
unclassified

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

and produces code very similar to subject code. Special attention is given to modeling student solutions by analogy, how students learn from doing, and how failures of working memory affect the course of problem solving. Of major concern is the process by which GRAPES compiles operators in solving one problem to facilitate the solution of later problems.

Abstract

We have gathered protocols of subjects in their first 30 hours of learning LISP. The processes by which subjects write LISP functions to meet problem specifications has been modeled in a simulation program called GRAPES(Goal Restricted Production System). The GRAPES system embodies the goal-restricted architecture for production systems as specified in the ACT* theory (Anderson, 1983). We compare our simulation to human protocols on a number of problems. GRAPES simulates the top-down, depth-first flow of control exhibited by subjects and produces code very similar to subject code. Special attention is given to modeling student solutions by analogy, how students learn from doing, and how failures of working memory affect the course of problem solving. Of major concern is the process by which GRAPES compiles operators in solving one problem to facilitate the solution of later problems.



Introduction

There are a number of reasons for being interested in how people learn to program. For one thing, it is an excellent example of the acquisition of a complex cognitive skill. It gets to the heart of many of the deep epistemological issues that haunt cognitive science. Particularly in the case of a language like LISP and in the case of subjects with no prior computer experience, we are looking at skill acquisition with relatively little relevant prior knowledge. It is almost as close to a situation of a tabula rasa as we are going to find in an adult. Also, in contrast to domains like natural language or perhaps even mathematics, it is extremely implausible to argue that we have evolved a special faculty for this skill (e.g., Chomsky, 1980). In studying the acquisition of programming skills, we are looking at an instance of learning by general-purpose mechanisms. Despite the relative paucity of prior knowledge and prior specialization, people can become quite proficient programmers. So the domain offers a testimony to the power of these general-purpose learning mechanisms.

Two of our three subjects also had no prior programming experience. Thus, we are looking at a particularly pure case of novel learning. However, the behavior we observe in LISP also shows up in learning other skills like geometry and physics and we feel our theoretical conclusions do generalize to other domains.

It is also the case that learning to program is going to be an increasingly important goal in our society. Thus, understanding its acquisition will have enormous educational impact. The issue of training novel, complex, and technical skills is a major one for our "high-tech" society with its need to retrain a large fraction of the work force. This retraining will not always be in programming, but in studying programming we are addressing issues important to many technical skills.

This paper reports research relevant to four major issues—how the problem solving involved in programming is organized, how the knowledge given in instruction is initially used to guide programming, how this knowledge becomes compiled so it can apply smoothly, and how working memory capacity limits the ability to program. While these are interesting questions in their own right, we are particularly interested in these questions because they are key to the ACT* theory of acquisition of cognitive skills (Anderson, 1982, 1983). Consonant with ACT* we will argue for four major conclusions:

(1) The problem-solving is organized hierarchically according to a set of goals and subgoals.

(2) The problem solving is largely guided at first by structural analogy to concrete cases.

(3) The ACT* processes of composition and proceduralization convert this knowledge into procedures specific to programming.

(4) Working memory impacts on behavior by affecting the probability of successfully executing the analogy and programming procedures.

The Data Base

We have looked extensively at the first 30 hours of novice programming behavior of three subjects (SS, WC, and BR). SS was an undergraduate with no prior programming experience, BR was a psychology BA also without any programming experience, WC was a college professor with some FORTRAN experience. In these protocols, subjects studied a text on LISP – SS studied Siklossy (1976), WC studied Winston (1977), and BR studied Winston & Horn (1981). We recorded their verbal protocols, kept their paperwork, and kept a record of their terminal interactions. The individual sessions varied from 45 minutes to two and a half hours, depending on what seemed to be natural units and natural breaking points. Approximately one quarter of the session time was spent reading and discussing the text; the other three-quarters of the time was spent doing various exercises. The subjects worked with an experimenter who tried to do as little teaching as possible and let the student learn from the text. The main responsibility of the experimenter was to query the subjects about what they were thinking, why they tried various solutions, etc. However, if the subject had a serious misunderstanding or was lost in problem, the experimenter would intervene with tutorial assistance.

We feel that we have a good record of the learning that was occurring in these sessions. Subjects were instructed not to think about LISP when they were not in the experimental session. They were also not permitted to keep the textbook between sessions.

While the 30 hour protocols from these subjects has been the major source of data for theory construction, we have also looked at protocols from these subjects much later, after they had continued their

LISP education; we looked at protocols from relatively advanced LISP programmers. In addition, we have assigned various learning problems to a large class learning LISP. While we cannot get any information about the real-time problem-solving from the class, the data does provide information about the distribution of final solutions. This provides one basis for judging the representativeness of the solutions we see from our three subjects.

The GRAPES Simulation

We developed GRAPES (Goal-Restricted Production System) to model how subjects write LISP functions[2] and how subjects learn from their problem-solving episodes. GRAPES is a simulation of certain aspects of the ACT* theory and as such takes the form of a production system. Each production in GRAPES has a condition which specifies a particular programming goal and various problem specifications. The action of the production can be to embellish the problem specification, to write or change LISP code, or to set new subgoals. A representative example of a production[3] that a pre-novice might have:

R1: IF the goal is to write a structure
and there is a template for writing the structure
THEN set a goal to map that template to the current case.

R1 might be invoked in a non-programming context such as when one uses another person's income tax form as a template to guide how to fill out his own. Productions like R1 serve as a basis for subjects' initial performance in LISP. A production that a novice might have after a few hours of learning is:

R2: IF the goal is to add List1 and List2
THEN write (APPEND List1 List2)

This production recognizes the applicability of the basic LISP function. With experience, subjects become more and more discriminate about when to apply LISP functions and more articulate about how to apply functions. A rule that an expert might have is:

R3: IF the goal is to check that a recursive call to a function will
terminate and the recursive call is in the context of a MAP function
THEN set as a subgoal to establish that the list provided to the MAP
function will always become NIL after some number of recursive calls

All programs in LISP take the form of functions that calculate various input-output relations. These functions can call other functions or themselves recursively. A programming problem is solved in GRAPES

by decomposing an initial goal of writing a function into subgoals and dividing these subgoals into others, etc., until goals are reached which correspond to things which can be directly written. The composition of goals into subgoals constitutes the AND-level of a goal tree; alternative ways of decomposing a goal constitute the OR level of the goal tree. The details of the GRAPES production system are described in Sauers and Farrell (1982). The architecture differs from other production systems (e.g. Anderson, 1976; Newell, 1973), primarily in the way it treats goals. At any point in time there is a single goal being focused upon and only productions relevant to that goal may apply. In this feature, GRAPES is like ACT* (Anderson, 1983) and other recent theories (Card, Moran & Newell,1983; Brown and Van Lehn, 1980). More generally, it is the case that GRAPES has a subset of the features contained in ACT*

The Transition to Behavior: The FIRST Problem

One of our consistent observations from the protocols is that subjects are not able to read instructions, of even modest complexity, and then generate, without error, the behavior instructed. This is not surprising given the ACT* theory. According to that theory, instructions are stored initially in a declarative form while behavior requires procedures which are represented as productions. Instructions cannot directly set up procedures to perform the skill. To get behavior, general interpretive productions must convert this knowledge into behavior. Many of the problems arise because of the indirection through these interpretive productions.

One of the problem-solving episodes of BR is typical of the difficulties people have in making the transition from instruction to experience. She had read the instruction on pages 33-37 of Winston and Horn on function definition and had turned to the first problem on page 37 to define the function FIRST which returned the first element of a list.

She extracted virtually nothing from the text instruction. What she did extract was a template for how to write a function definition:

```
(DEFUN <function name>
  (<parameter 1> <parameter 2>...<parameter n>)
  <process description>)
```

Winston and Horn assert "angle brackets delineate descriptions of things". She also studied three examples

of function definitions to which she referred while trying to write the function FIRST. One of these functions converted fahrenheit to centigrade:

```
(DEFUN F-TO-C (TEMP)
  (QUOTIENT (DIFFERENCE TEMP 32) 1.8))
```

The second exchanged the first two members of a list:

```
(DEFUN EXCHANGE (PAIR)
  (LIST (CADR PAIR) (CAR PAIR)))
```

The third returned the percentage by which the second argument is larger than the first:

```
(DEFUN INCREASE (X,Y)
  (QUOTIENT (TIMES 100.0 (DIFFERENCE Y X))X))
```

These are all referred to in BR's protocol which is given in Appendix A.

The one other relevant thing BR knew was the function CAR and how to use it when interacting with the monitor in LISP. CAR returns the first element of the list which is its argument. She knew, for instance, if she types (CAR '(A B C)), the monitor would return the answer A. Thus, this problem is really an exercise in using the syntax of function definition rather than an exercise in defining a novel function.

FIRST: The Protocol

Because raw protocols such as Appendix A are very complex and full of irrelevant detail, we have taken to simulating what we call *protocol schematics*. These are our intuitive characterizations of the essential features of the protocol, omitting many of the digressions. Table 1 provides a schematic protocol for the complete protocol given in Appendix A.

Insert Table 1 about here

There are two features of this protocol which are striking at the surface. First, as already noted, the subject makes very little use of the written instruction. She relies on the template for a LISP definition and the concrete examples. We use the term *structural analogy* to refer to the process by which subjects map such structures into new function definitions.

Second, at the end of the protocol, BR defined a function, SECOND, that returns the second element of a list. The striking feature of this episode is how much more rapidly the subject writes the definition for

SECOND than for FIRST. The subject does not have to resort to structural analogy and does not have difficulty with either feature that caused problems for FIRST – the specification of the parameter list or the specification of the LIST1 argument within the process specification. The only thing that is causing the subject any difficulty in SECOND is in deciding how to compose the primitive LISP functions CAR and CDR together[4]. The single experience with FIRST seems sufficient for her to compile rules about how to deal with many aspects of the syntax of function definition.

FIRST: The Simulation

We created a production system in GRAPES that would simulate this protocol. The only productions we required for this simulation were ones to do structural analogy and ones that could use the LISP functions CAR and CDR at the top level.[5] The first type of productions represent a general prior skill evoked in many contexts (for instance, in filling out income tax forms). The second type was acquired from work with earlier chapters in Winston and Horn.

Figure 1 illustrates the goal tree generated in simulating this example. Each box in Figure 1 represents a goal and each arrow emanating from the box represents a GRAPES production trying to achieve the goal. If the production has subgoals, it is connected to goal boxes below. The simulation starts with the goal of writing the function and chooses as its method to use the template for function definition as a model. We refer to this as *mapping* the template. The structural analogy productions respond to the goal of mapping the template by mapping the components DEFUN and <function name> in the template. These productions wrote out "(DEFUN FIRST" without difficulty. This gets us through line 3 of the schematic protocol in Table 1.

Insert Figure 1 about here

Like our subject, GRAPES was not able to directly write out the parameter part of the function template because GRAPES did not know what a parameter was. In cases like this, GRAPES' analogy productions will resort to a concrete example. The concrete example retrieved by GRAPES is the definition of F-TO-C given earlier. The subject in lines 5 and 6 reviews two examples in Winston and Horn – F-TO-C

and EXCHANGE. In both cases she notes the single argument which was the parameter. GRAPES solved the analogy: X is to F-TO-C as the parameter list (i.e., (<parameter 1><parameter 2>...<parameter n>)) is to the abstract template, and retrieved (TEMP) as the value for X. Thus, it decided (TEMP) was serving the parameter role in F-TO-C. Then it solved the analogy X is to FIRST as (TEMP) is to F-TO-C and determined the value for X was (LIST1) which it put into the function definition. That is, GRAPES decided (LIST1) served the same role in the function it was defining that (TEMP) was serving in F-TO-C. We infer that this is what the subject was doing in line 7 of the schematic protocol.

Then GRAPES turned to trying to map <process specification> from the template. Being unable to directly interpret what is meant by <process specification>, it looked to its concrete example F-TO-C to see that the LISP code which filled this slot performed the function operations. By analogy, GRAPES set its goal to write code that would perform the operations required by FIRST. The subject at this point (line 8) looked to a different example, INCREASE, for the same purpose of analogy. A GRAPES production for using CAR at the top level applied next (corresponding to line 9 of the protocol), but there was no production to specify how to write the argument to the CAR in the context of defining a function. GRAPES and the subject know that CAR will operate on LIST1, but they do not know the syntax for specifying LIST1. GRAPES again turns to its concrete example, F-TO-C and solves the analogy (CAR ARG) is to (QUOTIENT X) and retrieves (DIFFERENCE TEMP 32) as the value of X which is the first argument to QUOTIENT. It then solves the analogy problem of what it must do to LIST1 to make it like (DIFFERENCE TEMP 32) and decides it should embed LIST1 in parentheses. Similar to GRAPES, the subject on line 10 looks at a function EXCHANGE which has the same argument structure as F-TO-C. We assume she makes the same erroneous analogy because she writes (LIST1) on line 11. The first argument to QUOTIENT in F-TO-C is embedded in parentheses (as is the first argument to LIST in EXCHANGE) because a function, DIFFERENCE, must be called to calculate the argument for QUOTIENT whereas no embedded function call is required for the argument to CAR in the FIRST example.

There are two things to note at this point. First, the subject had read the information in the text that could have informed her to write LIST1 without parentheses, but this had no impact on her behavior.

Second, on previous occasions she had correctly specified variable arguments when evaluating functions at the top-level. Eventually, the tutor used this second fact, that the subject could do it correctly at the top level, to guide the subject to a correct solution. Both of these observations illustrate the relative isolation of knowledge. That is, knowledge studied or used in one context is not available in another context.

When the subject tries her function definition, an error is generated (line 14). GRAPES received the same error message when it tried out the same function definition that it generated. The error occurred because LISP treats the first thing inside a parenthesized expression to be evaluated as a function and there is no function corresponding to LIST1. GRAPES associated this error with the failure to correctly specify the argument to CAR. On previous occasions BR had encountered the same error at the top level typing in commands like (CAR (A)) where the argument (A),to CAR is to be taken literally rather than evaluated. Always in the past she had repaired these errors by *quoting* the argument. This is done by preceding the argument with a single quote, i.e. (CAR '(A)). We assume that both the subject and GRAPES have compiled from previous experience a rule that the way to repress this error is by using quote, which stops LISP from evaluating. Thus, both GRAPES and the subject generate the new function definition as it is given in line 15 of Table 1.

When this new function is tried on an example, LISP returns the CAR of '(LIST1) which is LIST1 – rather than the first element of the value of LIST1. It is at this point (lines 16 and 17) that the tutor intervenes and reminds the subject of how she would solve the problem at the top level. At the top level, the student would have used (CAR LIST1) rather than (CAR (LIST1)) or (CAR(LIST1)). We simulated this intervention in GRAPES by refocusing it to the code-first-relationship goal in Figure 1, and putting (CAR LIST1) as a top-level example in working memory. Then GRAPES, as the subject, maps this code to its current function definition and comes up with the correct code.

FIRST: Knowledge Compilation

After finally solving this problem, GRAPES formed two operators which aided its solution of the second problem. These operators summarized much of the problem solving that took place.

P1: IF the goal is to write a function of one variable

THEN write (DEFUN function (variable)
and set as a subgoal to code the relation calculated by
this function
and then write).

P2: IF the goal is to code an argument
and that argument corresponds to a variable of the function
THEN write the variable name.

We have encircled in Figure 1 the portions of the goal tree that are summarized by each of these productions. The first production captures the top level syntax of a function call while the second summarized the search involved in finding out how to specify a variable argument to a function. With these productions, GRAPES was able to write the function SECOND much easier, as was the subject.

We use *knowledge compilation* to refer to the process by which GRAPES forms such productions. Later in this paper we will describe the mechanisms underlying compilation in some detail; however, the examples above illustrate two important properties of compilation. First, in forming P1 it must be able to recognize which aspects of the process are variable and must be left as open subgoals. In forming P2, it must be able to recognize which parts of the goal tree were incorrect paths and which parts were critical to the final solution.

FIRST: Conclusions

There are a number of conclusions that we draw from BR's protocol and the GRAPES simulation. The first is the importance of structural analogy to bridging the gap between current knowledge and the needed behavior. We see two sources for the structure from which the analogy is being made. One is templates provided in the text and worked-out problems. The other is structures that the subject can generate—for instance, the subject generated (CAR LIST1) as a top-level solution and then used this in her function definition.

Bott (1978) and Rumelhart & Norman(1981) have also stressed the importance of analogy in early learning. In their situation, subjects were using analogy to extra-domain experiences. One feature of LISP is that there are very few relevant analogies to other domains. Therefore, the analogy process must use examples from within LISP.

A second conclusion concerns the hierarchical structure of the problem-solving episode as illustrated in Figure 1. Note that the goals in this tree are expanded depth-first, left-to-right (this mode of expansion is clearer in the next examples which involve "bushier" goal trees). Jeffries, Turner, Atwood and Polson (1981) also note this hierarchical, top-down, structure in the programming behavior of experts—although their subjects use breadth-first expansion in correspondence with the edicts of structured programming.

The third conclusion is the importance of knowledge compilation in extracting new production rules from an example problem. These rules streamline the solution of later problems. As the protocol shows, the learning can be on the basis of a single example. It needs to be stressed that the lessons of this example "stuck" which is to say, BR did not have, on later days, the same difficulty with the basic syntax of function definition nor argument specification. It should also be stressed that compilation depends critically on the structure of the goal tree being compiled.[6] That is, the structure of the goal tree identifies what parts of the problem-solving episode belong together and what can be collapsed into a single rule.

In these three features—structural analogy, hierarchical goal trees, and knowledge compilation—we have one complete solution to the issue of how the subject is able to make the transition to a new cognitive behavior. As such, they constitute a major conclusion of this paper.

Dealing With Gaps in Knowledge: The ONETWO Problem

We will provide a second example to help reinforce the conclusions from the first. This comes from the subject, SS, who was slightly more advanced at this point in her protocol. She had written the functions FIRST, SECOND, and THIRD. Thus, she had already learned the basics of function definition. She was then given a problem, ONETWO, that exposed some of the gaps in her knowledge. Thus, in this protocol, we will see how a subject deals with gaps in an existing procedure. This protocol is considerably more elaborate than the first. Whereas BR's protocol spanned a little less than half an hour, this protocol spanned about an hour. Rather than presenting it in its entirety, we have simply provided the protocol schematics in Table 2.

Insert Table 2 about here

The ONETWO problem required the subject to write a function which would take a list as an argument

and return a new list consisting of the first two elements of the argument list—e.g., (ONETWO '(A B C)) = (A B). The LISP functions that the subject knew at this time included CONS but the subject had not yet learned about LIST. CONS takes two arguments and inserts its first argument in the list that is its second argument, eg. (CONS 'A '(B C)) = (A B C). Although SS had never used CONS in a function definition she had a fair amount of experience with it at the top level of LISP when evaluating expressions such as the example above. She also had experience with one slightly esoteric fact which proved critical to the solution of the problem. If one used CONS with the second argument NIL, one puts the first argument in a list, eg. (CONS 'A NIL) = (A). This is because NIL is equivalent to an empty list, eg. NIL = ( ).

ONETWO: An Initial Attempt

Initially, the subject could not think of a plan for defining ONETWO, so the experimenter suggested writing a simpler function, ADDTWO, which would take two arguments and make a list out of them. SS was able to plan out a solution to ADDTWO much more easily. It is interesting to speculate why ADDTWO was more tractable than ONETWO. As we will see, the output specification and the basic solution did not change in going from ONETWO to ADDTWO. However, by reducing the complexity of the task by one level, the burden on the subject's working memory was reduced enough so that she was able to match rule conditions more easily.

Figures 2 - 7 illustrate the simulation's attempts to solve ONETWO. Given the close correspondence between the simulation and SS's protocol, we infer that these figures also describe the goal structures that were guiding her problem solutions.

Insert Figure 2 about here

Figure 2 illustrates the first work that was done on the ADDTWO subproblem. The first operator sets the subgoals of coding the function and checking(testing) the code. Unable to code the solution directly, GRAPES sets subgoals to come up with concrete examples of the input to ADDTWO and what its output should be, to find some code that could be used at the top level that would convert the concrete input into the concrete output, to check this code, and then to map this code into an abstract function definition. The inputs

SS chose to pass to ADDTWO were (A B) and (C D). Why she chose list arguments we are unsure. The result she wanted for these inputs was ((A B) (C D)). We constrained the GRAPES simulation to choose the same example.

Figure 3 illustrates the simulation of the process by which she decided what top-level code would mimic the performance of ADDTWO. After deciding on the example, she went through an episode where she explicitly reviewed the definition of all the functions she knew, searching for an appropriate one. She selected CONS, commenting that ADDTWO "is sort of like CONS except in CONS the first argument is any S-expression and the second argument is a list". We represented the definition of CONS in GRAPES as

> The first argument of CONS is any S-expression and the second argument is a list. Its result is a list. The first element of the resulting list is the first argument. The rest of the result consists of the second argument.

She and GRAPES chose CONS because they wanted a list and CONS makes lists. Having selected CONS, the subgoals were now to determine what arguments to pass to CONS in order to get the intended result.

Insert Figure 3 about here

The critical piece of information in selecting the first argument is the definition statement *The first element of the result is the first argument.* GRAPES interfaces this with the desired result, ((A B) (C D)), to determine that the correct argument should be (A B). Next, SS and GRAPES turn to the second argument. The appropriate part of this definition is *The rest of the result consists of the second argument.* Matching this would retrieve ((C D)) as the second argument. However, our subject retrieved (C D). We assume that the semantic features of *consists* were partially lost and this statement became *The rest of the result contains the second argument.* We manipulated GRAPES' working memory so that it would produce this error.[7] The subject and GRAPES mentally simulated what the outcome would be of the code (CONS '(A B) '(C D)). This involved retrieving the definition of CONS again. As evidence that her definition of CONS was not in error, she correctly determined that ((A B) C D) would result as an answer. On a few occasions in the past, SS had incorrectly used CONS at the top-level in just this way--having one less parenthesis around the second argument. Therefore, we assume she had compiled a rule to repair this which embedded the second

argument to CONS in an extra list. By applying this rule, she and GRAPES recover from their error and make up the concrete example (CONS '(A B) '((C D))).

To summarize, at this point the subject had actually created some LISP code which could be typed into the top level of LISP and was going to use the structure of this code to guide the creation of an abstract LISP function. This will be done by structural analogy or mapping. This mapping proceeds in basically the same way as the mapping of the function definition template by subject BR.

ONETWO: The Mapping

Insert Figure 4 about here

Figure 4 illustrates the simulation of SS's initial attempt to map from the concrete code, (CONS '(A B) '(C D)), to an abstract LISP function definition. GRAPES starts in that figure with the goal MAP TO ABSTRACT. First she maps CONS in the concrete code into CONS in the LISP function. At this point the structure of the function is:[8]

```
(defun addtwo (one two)
   (cons <?> <?>))
```

The remaining task is to map the two concrete arguments into abstract arguments. She first focuses on mapping (A B). The following rule, called MAP-FIND, applies:

```
IF the goal is to map an expression E into a domain D
   and E contains a term T
   and T corresponds to a argument A in domain D
THEN replace T in E by A
```

So, in this case she is trying to map the *expression* (A B) to the *domain* of the function definition where the *argument* ONE in the function definition corresponds to the *term* (A B)—in this case the term is the whole expression. Therefore, after replacing the argument for the term, the expression becomes simply ONE. This same rule applies to map the second *concrete expression* ((C D)). In this case the argument TWO corresponds to the *term* (C D) and the expression after substitution is (TWO). Note this rule has mapped the first concrete expression into a correct definition expression but has mapped the second concrete expression into an incorrect definition expression. The function definition at this point is:

```
(defun addtwo (one two)
   (cons one (two)))
```

Insert Figure 5 about here

Figure 5 illustrates some of the subsequent evolution of this definition. The coding of ADDTWO had the brother goal of checking that code. Both SS and GRAPES called the LISP interpreter to try the code with the arguments (A B) and (C D)--i.e., both evaluated (ADDTWO '(A B) '(C D)). Both received the same error message "TWO undefined function object." This corresponds to an error that SS had encountered a few times previously in her problem solving. In previous occasions, the cause had been failure to quote an argument. Therefore, we assumed that she had compiled an operator that used quote to stop evaluation.[9] When this operator applied, her LISP code became

```
(defun addtwo (one two)
   (cons one '(two)))
```

Again, she tried the code. This time it returned the result ((A B) TWO). Comparing this with her desired result the problem was localized to the second argument given to CONS; she and GRAPES went back to retrying the goal of mapping ((C D)).

Insert Figure 6 about here

Figure 6 illustrates the simulation of this mapping. Having returned to this goal, the previous MAP-FIND operator will not apply again. Therefore, a default rule applies which creates a new subgoal of coding a list consisting of a single argument.[10] As in the case of coding the full ADDTWO problem, GRAPES falls back on the plan of making up a concrete example, coding it, checking the code, and then mapping the code into an abstract code for the function. The previous concrete example of ((C D)) is used. Again, CONS is chosen because it makes lists and again its definition is used to determine the correct arguments. This time the definition is correctly used and GRAPES plans the concrete code as (CONS '(C D) NIL).

After checking this code, GRAPES turns to the goal of mapping the concrete code to the LISP function. The process of performing this mapping is quite analogous to the original mapping in Figure 5. Again,

CONS is mapped into CONS. The same MAP-FIND operator as before maps (C D) into TWO. An operator for special LISP symbols, like NIL, maps NIL onto itself. So, the final successful code becomes:

```
(defun addtwo (one two)
    (cons one (cons two nil)))
```

One interesting feature of this example is that SS is able to find her way eventually tc the correct function without ever correcting the MAP-FIND operator, which will erroneously apply whenever it is given a non-atomic data structure. Later protocols by SS indicated she still had the erroneous MAP-FIND operator. An examination of buggy functions submitted by students given class exercises suggests that this is a frequent bug among LISP novices.

ONETWO: Return to the Main Function

Figure 7 illustrates the behavior of the simulation and the subject when they returned to the original ONETWO problem. The code they generated is given below:

```
(defun onetwo (list)
   (cons (first list)
     (cons (second list) nil)))
```

Whereas the subject had taken an hour to code ADDTWO, she only took ten minutes to solve ONETWO and most of that time was spent confirming what the functions FIRST and SECOND did. ONETWO is solved by the same method that ADDTWO is solved, but without any rehearsal of the ADDTWO method, nor any of the use of examples that was such a large part of the ADDTWO solution. Our assumption is that operators were compiled from the ADDTWO problem that summarized the planning steps and these operators facilitated solution of the ONETWO problem.

Insert Figure 7 about here

One of the operators that GRAPES compiled summarizes the problem solution illustrated in Figure 6. In creating the operator, GRAPES must distill those aspects critical to the solution. The goal in Figure 6 was to create a list of a single element and this was eventually achieved by the action of CONSing that element with NIL. Most of the intermediate results in Figure 6 were not part of this final solution and can be deleted in the compiled production. We will shortly discuss how compilation achieves this. The summary operator

built is:

IF the goal is to code a list consisting of one argument
THEN CONS that argument with NIL
and set as a subgoal to code that argument

Similarly, an operator is compiled to correspond to the outer CONS in the ADDTWO function. It has the form:

IF the goal is to code a list consisting of argument1 and argument2
THEN CONS argument1 into a list consisting of argument2
and set as subgoals to code argument1
and to code a list consisting of argument2

Many other operators are compiled which are less useful. These other operators are not harmful, they are just too large or too specific to apply in future situations.

ONETWO: Summary

This examination of ONETWO reinforces some of the conclusions from the first protocol and simulation. Again we see the use of structural analogy. In this case the subject did not take her analog from the text but rather generated a concrete example of LISP code at the top level that could serve as an analog. Second, the hierarchical structure of the problem-solving is even clearer in this simulation because of its greater complexity. Third, we see the importance of knowledge compilation in building new operators that will summarize the lessons learned from one problem-solving episode. We see one instance of an additional phenomenon that will loom larger in the third and forthcoming simulation and protocol. This is the episode where the subject temporarily forgot the definition of CONS. Such memory failures can become a dominant feature of some problem-solving episodes.

Further Discussion of Compilation

As discussed in Anderson (1982) there are two components to compilation— proceduralization and composition. Proceduralization refers to the creation of specific productions that eliminate retrieval of information from long-term memory by building that information into the rule. Composition refers to the creation of more efficient productions that take the place of several productions. Both components were involved when compiling the operators in the ONETWO example, but there are other circumstances where the two might operate singly.

Proceduralization

Proceduralization can be illustrated in its pure form by the following example: in GRAPES there is a production that will retrieve function definitions from long-term memory and apply them:

IF the goal is to code a relation defined on an argument
and there is a LISP function that codes this relation
THEN use this function with the argument
and set as a subgoal to code the argument

In this production, *relation* and *function* are variables which allow the production to match different data. The second line of the condition might match, for instance, "CAR codes the first member of a list." If this rule is proceduralized to eliminate the retrieval of the CAR definition, it becomes

IF the goal is to code the first member of a list
THEN use CAR of the list
and set as a subgoal to code the list

This is achieved by deleting the second clause in the first production that required long term memory retrieval. In addition, the rest of the production is made specific to the relation *first element* and the function *CAR*. Now a production has been created which can directly recognize the application of CAR. This will result in a reduction in the amount of long-term memory information that needs to be maintained in working memory.

Composition

As an example of pure composition, suppose one wanted to add the first member of List1 to List2. Then the following two operators would apply in sequence:

IF the goal is to add an element to a list
THEN use CONS on the element and the list
and set as subgoals to code the element
and to code the list

IF the goal is code the first member of a list
THEN use CAR on the list
and set as a subgoal to code the list

The first rule above would apply binding *an element* to "the first member of List1" and *a list* to "List2". The second production would apply binding *a list* to "List1". A simple case of composition would involve combining these two productions together to produce

IF the goal is to add the first member of one list to another list

THEN CONS the CAR of the first list to the second list
and set as subgoals to code the first list
and to code the second list

Such composition would collapse repeated sequences of coding operations to create macro-operators. The result would be a speed-up in coding. The technical issues of how to combine productions together are fairly straight forward and are discussed in Anderson (1983) and Neves & Anderson (1981). A major issue concerns what productions to compose together. The above example is a fairly simple case of collapsing two levels of a goal tree into one. However, in some cases such as when Figure 6 was collapsed into a single production many productions are collapsed. GRAPES determines what productions to collapse by inspecting the goal tree. There are two types of goals for purposes of composition: *inherent goals* and *planning goals.* Inherent goals are intrinsic parts of the programming task. For current purposes inherent goals are all variants of writing code. The important feature of inherent goals is that, in achieving them, one achieves part of the original task. On the other hand, planning goals produce results that are used to guide solution of the original problem but the results themselves are not part of the final solution. In Figure 7 the inherent goals are "CODE LIST OF ONE ELEMENT" and "CHECK TWO"; all the rest are planning goals.

Composition collapses productions in one of two ways. One way is to eliminate the planning goals that are intermediate between two inherent goals. This is what happens in Figure 7. In doing this it is compiling out the planning process and simply leaving in the products of that planning. The second possibility is illustrated in the case above. Here it skips over the setting of an intermediate inherent goal and so reduces the number of inherent goals by one. In doing this it is basically creating macro operators somewhat similar to STRIPS(Fikes & Nilsson, 1971) This learning scheme requires that the learner be able to identify what subgoals were essential to the problem solution and which are only intermediate to the final solution.

It needs to be emphasized that neither proceduralization nor composition eliminate the original production rules from which they were built. Rather the new compiled rules just serve as additional supplemental rules to produce better performance in certain circumstances.

The effect of the knowledge compilation process is to create a set of productions that mirror the

structure of LISP. They may explicitly involve LISP functions like CAR and COND or LISP programming techniques like CDR-recursion (see forthcoming discussion of POWERSET). These productions will preserve the inherent goals which are specific to LISP and will delete the planning goals involved in domain-general processes like structural analogy. Thus representative productions become (see Anderson, Sauers, and Farrell, 1982):

P1: IF the goal is to code the second member of a list
THEN use CADR and set a subgoal
to code the list.

P2: IF the goal is to obtain all the elements which have
a relation to any member of a list
THEN use MAPCONC and set as subgoals
1. To code a function that will return all the elements that have
a relation to the argument.
2. To code the list.

The programming behavior we see in GRAPES once such productions are acquired, is somewhat like the PECOS system of Barstow (1979). As discussed in Anderson, Sauers, and Farrell (1982), the main programming activity for the more advanced student becomes algorithm design (see Kant and Newell, 1982) in which the task is to convert the problem specification into a form that such rules can apply.

The expert programmer is advanced over the novice both in possession of rules for reformulating problems and rules which associate large templates of code with specific problems. This is an idea that has been suggested by a number of researchers (Kahney & Eisenstadt, 1982; Soloway, 1980;Rich & Shrobe, 1978). Many of these rules are explicitly learned either through formal courses or informal interaction with other programmers. However, we suspect that many more are also compiled from experience. That is, the programmer hits upon a problem, solves it with much search and effort, and compiles a rule that captures the essence of the solution.

An Advanced Problem: POWERSET

After 30 hours of learning, subjects are beginning to solve relatively complex problems although they are still novices. Consider a problem that was solved by all three of our subjects (for reports of other advanced problems see Anderson, Farrell, and Sauers, 1982). We will describe WC's solution to this problem as it was the most straightforward solution that we got from the subjects. In looking at this problem we will

see a case where the student has learned all the basics and the main problem is putting these basics together. As argued earlier, knowledge compilation creates a system of productions whose control structure mirrors the structure of LISP. Although WC and other subjects possessed all of the necessary tools to solve the problems they are given, their solutions are often riddled with working memory failures.

The problem is called POWERSET and Figure 8 illustrates how it was presented to the subjects. The subject is told that a list of atoms encodes a set of elements and he is to calculate the powerset of that set—that is, the list of all sublists of the original list, including the original list and NIL. Each subject was given an example of the POWERSET of a three element list. The three subjects we observed in detail spent from under two hours to over four hours solving this problem. In each case, they spent about one-third of their time uncovering a key insight and the other two-thirds of their time working out the LISP code that would capitalize on this insight.

Insert Figure 8 about here

We have also assigned this problem to a number of programming classes and gathered informal problem solution reports. There are two types of solutions which subjects are prone to attempt and which tend to distract them from the correct insight:

1. There is a strong tendency to try to implement the way they would solve the problem by hand. For most subjects this hand solution is one in which they calculate the null list, then all the singleton lists, then all the doubleton lists, etc. (i.e. NIL, the (A),(B),(C), and then (A B), (B C), etc.)

2. Some subjects are distracted by the fact that certain sublists can be achieved quite easily by taking CDR's. So, given the example (A B C), the sublists (B C), (C), and ( ) can be gotten by taking successive CDR's. This leaves the difficult task of calculating the non-CDR's.

However, almost all subjects finally come up with basically the same solution. The prototypical solution to the problem is given in Table 3. The essential insight is illustrated in Figure 9. This involves noticing the

relationship between the POWERSET on the full list and POWERSET on the tail (CDR) of the list. In Figure 9 we denote by X the result of POWERSET on the full list and we denote by Y the result of POWERSET on the tail of the list. Subjects noted that Y provided half of the members they would need for X. Second, they noted that the other half could be gotten from Y by adding A, the first member of the list L, to each member of Y. Thus, X is formed from the lists Y and Z, where Z is formed from Y by adding the first member of L to each member of Y.

Insert Table 3 and Figure 9 about here

The decision to consider the relationship between (POWERSET L) and (POWERSET (CDR L)) is not just a stab in the dark. It is dictated by a recursive programming technique that the students were taught called CDR-recursion. This technique involves assuming that the function will return the correct result for the CDR of the list and trying to use this result to calculate the correct answer for the whole list.

POWERSET: Simulation of WC

WC took slightly under two hours to solve the problem, of which the first half hour was spent formulating the critical insight. His schematic protocol is given in Table 4. He spent some time trying to formulate the solution by taking successive CDR's before he abandoned this effort. The critical point in his protocol came when he decided to examine the relationship between the powerset of the original list and its CDR (lines 8 and 9 in Table 4).

Insert Table 4 about here

It would be interesting to try to simulate the process by which the subject comes to the insight about the relationship between the powerset of the whole list and the powerset of the tail of the list – i.e., $X = Y + Z$[11]. This would begin to get us into issues of algorithm design and that is beyond the scope of the current report. However, we will focus on the programming that is involved in converting this insight into LISP code.

Figure 10 illustrates GRAPES' goal structure for this problem. GRAPES keys off the fact that the argument is a list to attempt the CDR-recursion technique. This technique involves two subgoals. One is to

write the code for the recursive step and the other is to write the code for the terminating step which is when the argument to POWERSET is the empty list, NIL. Under the recursive step, there are two subgoals. One is to characterize the relationship between POWERSET of the full list and POWERSET of the tail of the list. The other is to convert that characterization into LISP code. Not trying to simulate the insight X = Y + Z we simply provide GRAPES with this information outright.

Insert Figure 10 about here

Both WC and GRAPES turn to coding Z which is formed from Y by adding *A to each member of Z*. Since GRAPES knows no function that will calculate such a relation, it sets out to write a new function ADDTO that will calculate this relation. WC however, in line 10 of Table 4, first writes (UNION (CAR L) (POWERSET (CDR L))). UNION is a function which combines two lists and avoids repeats. This clearly will not give Z. It seems he has a vague specification in working memory of combining A with Y and UNION matches this specification on the basis of it being a combining function. WC knows quite well what UNION does and as evidence of this, he corrects his code a couple of minutes later, and articulates what is wrong without intervention of the experimenter.

POWERSET: Coding of ADDTO

Continuing with the depth-first goal expansion, both GRAPES and WC turn to writing the ADDTO function before completing POWERSET. The goal structure for ADDTO is illustrated in Figure 11. The function is written with the same cdr-recursion technique used in POWERSET. More advanced students might recognize this as basically a simple iterative structure and solve it with a PROG or MAP, but we are simulating WC at the point where he has not been taught about PROG's or MAP's and only knows about recursion within LISP and not iteration.

Insert Figure 11 about here

When WC first turned to coding the recursive step he wrote (CONS (LIST A (CAR L))(ADDTO A (CDR L))). This differs from the correct code in that the function LIST is used rather than CONS. Rather

than combining A and (B C) to get (A B C), this will combine them to get (A (B C)). Once again our subject confuses two similar functions; in this case LIST, which makes its arguments elements of a list, is confused with CONS, which inserts its first argument into the list which is its second argument. This is all the more interesting because this line of code also contains a correct use of CONS. It needs to be stressed that WC knows quite well the distinction between CONS and LIST.

Then WC turned to writing the appropriate code for the terminating condition—i.e., when ADDTO is called with arguments A and NIL. His first thought was that he should add A to this empty list and return (A). That is, he had lost sight of the fact that the second argument to ADDTO is a list of lists and he should add A to each sublist. This is another example of the subject losing track of what it is that he had intended to do. The subject discovered the problem with this code by mental simulation and put in the correct terminating value, namely, NIL.

At this point, the buggy definition of ADDTO was typed into the terminal and tried it out on some sample problems. By tracing the function, he spotted and diagnosed the problem caused by his use of LIST rather than CONS. He changed this and the function ran correctly. WC corrected his problem without instruction from the experimenter and without looking up CONS or LIST in his text.

POWERSET: Return to Main Function

Having completed ADDTO, he then returned to writing POWERSET. His first remark in line 18 indicates that he had completely forgotten the series of goals that led to ADDTO. He had to re-read the code he had written to reconstruct his goals.

After he reconstructed his plan for POWERSET, WC turned to coding the terminating condition. His first inclination was to return NIL as the value when POWERSET was called with the argument NIL. This was the only place that the experimenter intervened with some suggestions. He pointed out that the POWERSET is defined as the set of all subsets of a set. Any set is considered a subset of itself and therefore the set itself should be in the powerset. The experimenter explained that among the elements of the powerset of the empty set should be the empty set itself. From this explanation, WC inferred that the result for

POWERSET of NIL should be (NIL) rather than NIL. WC wrote (LIST NIL) but commented that he really did not understand the explanation.

Then the function was typed into the terminal and WC watched it run with a trace on POWERSET. When he saw POWERSET return (NIL) for the value of NIL and when he saw how this result was used by higher levels of POWERSET, he remarked that he now understood why (NIL) was the right value for the terminating condition. He still did not understand the experimenter's logical argument but he had an understanding of why the result was essential to the correct working of the function.

There is a close correspondence between WC and GRAPES in the overall flow of control among goals created by the decomposition strategy. However, there are frequent failures of memory on WC's part which are not part of the simulation. He loses track of both partial products calculated in the course of planning a function and incorrectly retrieves functions from memory. We have observed a similar high frequency of errors in all our novice subjects. Such errors are less frequent with advanced LISP programmers when they work on problems like POWERSET. Also, errors like the LIST-CONS confusion are almost non-existent when subjects are asked to execute a command at the top-level of LISP. They only appear embedded in the context of a problem with considerable working memory load. (A recent experiment conducted on a class of 60 novice programmers has confirmed that LIST-CONS confusions are more common when the function use is embedded within the other functions (See Anderson, 1983b)).

Analysis of Retrieval Failures

Working memory failures are the cause of certain problems in the protocol like forgetting why ADDTO was written. We think working memory failures are also responsible for the incorrect retrievals of functions like UNION and LIST.

The following is our analysis of the LIST-CONS confusion. It is similar to what Norman (1981) called a description error. We assume that the subject represents as his goal

1. To create a LIST L
2. where the first element of L is A

3. and where the rest of the list *contains* the elements of B.

This matches the specification of CONS. On the other hand, if the third clause above had contains the elements of replaced by simply contains, then it would match the specifications of LIST. If we assume that the relation contains is simpler than contains the elements of and involves a subset of its semantic features, we would predict that subjects would tend to lose the distinguishing features under heavy memory load and retrieve LIST instead of CONS. Also, this analysis would predict that CONS should not be intruded instead of LIST. This asymmetry is clearly the case in our protocols. The asymmetry has been shown to be statistically reliable in large-scale class experiments as well.[12] This analysis is also consistent with a different CONS error that we saw in the SS protocol involving ONETWO.

The basic analysis of working memory errors being offered here is one in which the initial representation is fragmented into a simpler representation. This is similar to the fragmentation forgetting theory of Jones (1976). As the above analysis of the LIST-CONS confusion illustrates, the result of such fragmentation depends critically on one's representation of the initial knowledge structure. Thus, one can use the nature of these working memory errors to make inferences about the knowledge representation such as the above analysis of the difference between CONS and LIST.

General Conclusions

While we do not have a complete theory of the behavior that occurs in the first 30 hours of programming, we have gained some interesting insights. The first has to do with the basis for the programming skill. Initially, the behavior is guided heavily by structural analogy in which the subject uses the structure of definitions and examples to guide the programming. Later productions are compiled which directly reflect the structure of the domain. We discussed how operators become compiled to reflect the logical structure of the basic LISP functions. In addition, the subject learns various techniques for transforming problems into formulations that can be encoded. For instance, WC's use of the CDR-recursion technique was critical to his solution of the POWERSET problem.

Second, whether the subject is using structural analogy or using domain-specific operators, the structure of the solution tends to be hierarchical, top-down, and depth first. This is strong support for the architectural

principle of goal-structured productions in the design of ACT*. It is also consistent with the view of problem-solving developed in Sacerdoti (1977).

Third, we have identified the learning mechanism of knowledge compilation as critical to the transitions underlying the learning process. We showed that GRAPES' compilation applied to the solution of one problem produced the improvement observed in its solution to the next problem. One of compilations' striking features is that it appears to occur so rapidly. We saw two examples where subjects learned from single problem-solving episodes. Such rapid compilation has also been observed in our analysis of geometry problem-solving(Anderson, 1982).[13] Finally, we think working-memory limitations become increasingly important as the novice learns the basics of LISP. In the terms of Norman and Bobrow (1975) the novices initial problem-solving is data-limited, but it rapidly becomes resource-limited and one important resource is working memory. Anderson (1982) argued that the major factor limiting rate of learning is working memory capacity. Elsewhere (Anderson, 1983b) it has been argued that working memory actually increases in its capacity to hold information about the domain, but there is nothing in our data on LISP to distinguish this hypothesis from the idea that subjects just develop more efficient coding schemes (e.g., chunking). For other domains, Chase and Ericsson (1983) provide fairly convincing arguments for increased working memory capacity.

By the way of summary, the following is the general characterization that we would like to give of learning to program in LISP. The students start out with various templates and examples and a set of facts that guide analogical use of these templates. With experience, analogy drops out and operators specific to LISP appear. Further improvement in LISP is strongly controlled by working memory capacity.

References

Anderson, J.R. Language, Memory and Thought. Hillsdale, NJ: Erlbaum, 1976.

Anderson, J.R. Acquisition of cognitive skill. Psychological Review, 1982, 89, 369-406.

Anderson, J.R. The Architecture of Cognition, Harvard University Press, 1983.

Anderson, J.R. Learning to Program. In the Proceedings of the Eighth International Joint Conference on Artificial Intelligence, 1983b.

Anderson, J.R., Farrell, R., & Sauers, R. Learning to Plan in LISP, ONR Technical Report, ONR-82-2, Carnegie-Mellon University, 1982.

Barstow, D.R. An experiment on knowledge-based automatic programming, Artificial Intelligence, 1979, 12, 73-119.

Bott, R.A. A study of complex learning, theory, and methodologies. Unpublished doctoral dissertation. University of California, San Diego, 1978.

Brooks, R.E. A model of human cognitive behavior in writing code for computer programs. Unpublished doctoral dissertation, Carnegie-Mellon University, 1975.

Brown, J.S. & Van Lehn, K. Repair theory: A generative theory of bugs in procedural skills. Cognitive Science, 1980, 4, 379-426.

Card.S.K., Moran, T.P. & Newell, A. The Psychology of Human-Computer Interaction. Hillsdale, N.J.: Erlbaum, 1983.

Chase, W. G. and Ericsson, K. A. Skilled memory. In J.R. Anderson (Ed) Cognitive Skills and Their Acquisition. Hillsdale, NJ: Erlbaum, 1981.

Chomsky, N. Rules and representations. Behavioral and Brain Sciences, 1980, 3, 1-61.

Fikes, R.E. & Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. Artificial Intelligence, 1971, 2, 189-208.

Jeffries, R., Turner, A.A., Polson, P.G., & Atwood, M.E. The processes involved in designing software. In J.R. Anderson (Ed.), Cognitive Skills and Their Acquisition. Hillsdale, NJ: Erlbaum, 1981.

Jones, G.V. A fragmentation hypothesis of memory: Cued recall of pictures and sequential position. Journal of Experimental Psychology: General, 1976, 1, 277-293.

Kant, E. and Newell, A. Problem solving techniques for the design of algorithms. In the Proceedings of the Symposium on the empirical foundations of information and software science. Atlanta, GA, November, 1982.

Kahney, H. & Eisenstadt, M. Programmers' mental models of their programming tasks: The interaction of real-word knowledge and programming knowledge. Proceedings of the Fourth Annual Conference of the Cognitive Science Society, 1982.

Neves, D.M. & Anderson, J.R. Knowledge Compilation: Mechanism for the automatization of cognitive skill. In J.R. Anderson (Ed.), Cognitive Skills and their Acquisition. Hillsdale, N.J.: Lawrence Erlbaum Associates, 1981.

Newell, A. Production Systems: Models of control structures. In W.G. Chase (Ed.), Visual Information Processing. New York: Academic Press, 1973.

Norman, D.A. Categorization of action slips. Psychological Review, 1981, 88, 1-15.

Norman, D.A. & Bobrow, D.G. On data-limited and resource-limited processes. Cognitive Psychology, 1975, 7, 44-64.

Rich, C. & Shrobe, H. Initial report on a LISP programmers' apprentice. IEEE Trans. Soft. Eng., SE-4:6, 1978, 456-466.

Rumelhart, D.E. & Norman, D.A. Analogical processes in learning. In J.R. Anderson (Ed.), Cognitive Skills and Their Acquisition. Hillsdale, N.J.: Erlbaum, 1981.

Sacerdoti, E.D. A structure for plans and behavior. New York: Elsevier, North Holland, 1977.

Sauers R., & Farrell, R. GRAPES User's Manual. Technical Report ONR-82-3.

Siklossy, L. Let's Talk LISP. Englewood Cliffs, NJ: Prentice-Hall, 1976.

Soloway, E.M. From problems to programs via plans: The context and structure of knowledge for introductory LISP programming. Coins Technical Report 80-19, University of Massachusetts at Amherst, 1980.

Winston, P.H. Artificial Intelligence. Reading, MA: Addison-Wesley, 1977.

Winston, P.H. & Horn, B.K.P. LISP. Reading, MA: Addison-Wesley, 1981.

Table 1
Schematic Protocol for BR
Writing FIRST and SECOND

1. Subject reviews template for function definition.

2. Subject reads problem.

3. Subject writes out '(DEFUN FIRST'.

4. Subject is confused by "parameters" in the definition template.

5. Subject reviews F-to-C and notes TEMP is the parameter.

6. Subject reviews the parameter EXCHANGE and notes PAIR is the parameter.

7. Subject decides LIST1 is the parameter for FIRST and writes (LIST1).

8. Subject looks at INCREASE.

9. Subject decides to use CAR.

10. Subject looks at EXCHANGE.

11. Subject writes (CAR (LIST1)).

12. Subject balances parenthesis. The function is:

```
(DEFUN FIRST (LIST1)
  (CAR (LIST1))).
```

13. Subject tries (FIRST '(B R)).

14. Subject reads error message "Error: Eval: undefined function LIST1".

15. Subject tries to insert a quote to prevent LIST1 from being treated as a function. The new definition is:

```
(DEFUN FIRST (LIST1)
  (CAR '(LIST1))).
```

16. Subject tries (FIRST '(B R)) again and reads the answer LIST1.

17. Subject claims not to know what to do. Tutor intercedes with a top-level example. She types (SETQ LIST1 '(B R)) and asks subject to get the CAR of (B R) using LIST1.

18. Subject writes (CAR LIST1).

19. Subject notes difference between what she just wrote and what she wrote in the function definition (CAR (LIST1)). Subject decides to replace the code in the function definition by what she has just written.

20. Subject decides she does not need a SETQ in the function definition. Subjects definition now is:

```
(DEFUN FIRST (LIST1)
  (CAR LIST1)).
```

21. Function FIRST works.

22. Subject reads specification of function SECOND.

23. Subject writes (DEFUN SECOND (LIST1).

24. Subject decides CDR will take her to the second element and CAR will extract it.

25. Subject reviews a previous top-level example, (CAR (CDR '(A B C))).

26. Subject finishes function definition which is

```
(DEFUN SECOND (LIST1)
  (CAR (CDR LIST1))).
```

27. Subject tests out definition on an example and it works.

Table 2
Schematic Protocol for SS Solving ONETWO

1. Tutor suggests writing a function ONETWO that returns a list of the first two elements of the input lists.

2. Subject reviews all of the functions that she has learned and comments that CONS seems that it might be useful.

3. Subject is stuck.

4. Tutor intervenes to suggest writing an easier function, ADDTWO, that takes two arguments and makes a list out of these.

5. Subject comments that this is like CONS and reviews definition of CONS.

6. Subject considers a concrete example, (ADDTWO '(A B) '(C D)) = ((A B)(C D)). She considers whether there is some way of converting (A B) and (C D) into ((A B)(C D)).

7. Subject suggests trying (CONS '(A B) '(C D)) but notes that this will produce ((A B) C D).

8. Subject changes code to (CONS '(A B) '((C D))).

9. Subject starts to write function definition – (DEFUN ADDTWO (ONE TWO).

10. Subject first writes (CONS ONE TWO) but then changes this to (CONS ONE (TWO)). The definition now is

```
(DEFUN ADDTWO (ONE TWO)
  (CONS ONE (TWO))).
```

11. Subject tries function definition with arguments (A B) and (C D). She receives the error message "TWO undefined function object".

12. Subject corrects by quoting. Function definition now is

```
(DEFUN ADDTWO (ONE TWO)
  (CONS ONE '(TWO))).
```

13. Subject tries new definition with argument (A B) and (C D). The result is ((A B) TWO).

14. The tutor tries to explain evaluation but has no success in getting her to correct the code.

15. The tutor suggests a subproblem of getting a list containing the second argument.

16. Subject suggests using CONS on the argument and NIL.

17. Function definition now is

```
(DEFUN ADDTWO (ONE TWO)
  (CONS ONE (CONS TWO NIL))).
```

18. Subject tries function definition out on (A B) and (C D) and the result is ((A B) (C D)) which is what is wanted.

19. Subject returns to the goal of writing ONETWO.

20. Subject writes (DEFUN ONETWO (LIS) (CONS (FIRST LIS).

21. Subject interrupts function definition to check what FIRST does.

22. Subject completes code. The function definition is

```
(DEFUN ONETWO (LIS)
  (CONS (FIRST LIS) (CONS (SECOND LIS) NIL))).
```

23. Subject tries definition with argument (A B C) and it returns the answer (A B) which is correct.

24. Tutor asks her to redefine ONETWO using ADDTWO.

25. Subject writes

```
(DEFUN ONETWO (LIS)
  (ADDTWO (FIRST LIS) (SECOND LIS))).
```

26. Subject tries definition with argument (A B C) and it returns the correct answer (A B).

Table 3
Prototypical Solution to POWERSET

```
(DEFUN POWERSET (L)
   (COND ((NULL L) (LIST NIL))
         (T (APPEND (POWERSET (CDR L))
              (ADDTO (CAR L) (POWERSET (CDR L)))))))

(DEFUN ADDTO (A Y)
   (COND ((NULL Y) NIL)
         (T (CONS (CONS A (CAR Y))
              (ADDTO A (CDR Y))))))
```

Table 4
Schematic Protocol of WC on POWERSET

1. The experimenter explains the problem.

2. The subject recognizes that there are 2^n sets in the solution where n is the length of the list.

3. The subject thinks about "taking the first element off of the list and calling this function on the rest of the list" because that is "the general thing I have been doing lately with recursion."

4. Subject switches attention to calculating all the cdrs of the list.

5. Subject now suggests a loop in which he successively takes all the powersets of successive sublists of the original list—"For every list I pull off an element and do powerset of the cdr."

6. Experimenter suggests the method will not work.

7. Subject focuses on powerset of (A B C D) and decides his method would "miss the sets with A in it."

8. Subject figures out the powerset of (B C D) and notes "All that I am missing is the union of A with all these things."

9. Subject states his plan "I have the powerset of (B C D) and I want to UNION that with something else which is A added to the powerset."

10. Subject writes

```
(UNION (POWERSET (CDR X))
   (UNION (CAR X) (POWERSET (CDR X)))).
```

11. Subject decides the embedded UNION will not work and decides to write a helping function called ADDTO with arguments A and L.

12. Subject decides he will add A to each member of L by CONS.

13. Subject decides when L is NIL he will add A to NIL to get (A). Then he changes his mind and decides NIL is the correct answer in this case.

14. Subject writes

```
(DEFUN ADDTO (X L)
   (COND ((NOT L) NIL)
         (T (CONS (LIST X (CAR L))
                  (ADDTO X (CDR L)))))).
```

15. Subject tries function (ADDTO 'A '((B C) ((D) (A B C D E)))) and traces.

16. Inspecting trace he decides he should replace LIST by CONS.

17. Corrected function runs correctly.

18. Subject comments "Now I have forgotten where I was in this thing." He reviews what he had written about POWERSET.

19. Subject writes

```
(DEFUN POWERSET (L)
  (COND ((NOT L)
       (T (UNION (POWERSET (CDR L))
            (ADDTO (CAR L) (POWERSET (CDR L)))))).
```

20. Turns to case when L is NIL and comments "I think I want to return NIL."

21. Types in (POWERSET '(A B)) and traces. He focuses on why POWERSET of (B) did not work.

22. Subject decides that problem is in ADDTO and he should correct it so ADDTO (B NIL) = (B).

23. Tutor tells subject to work on POWERSET and explains that the POWERSET of the empty set is a set that contains the empty set.

24. Subject corrects POWERSET so it now reads

```
(DEFUN POWERSET (L)
  (COND ((NOT L) (LIST NIL))
       (T (UNION (POWERSET (CDR L))
            (ADDTO (CAR L) (POWERSET (CDR L)))))).
```

but comments he does not understand the terminating condition.

25. The subject tries the function in an example with a trace.

26. The subject comments on the trace "Oh, I think I understand it now. It was returning NIL rather than a list with NIL in it...Well I think I understand why this didn't work that time. I still don't understand the way you think about it. But I think the way I think about it is OK too."

Figure Captions

Figure 1 A representation of the goal structure in subject BR's solution to the problem of writing the function FIRST. The boxes represent goals and the arrows indicate that a production has decomposed the goal above into the subgoals below. Checks indicate successful goals and X's indicate failed goals. The dotted lines indicate parts of the goal tree combined in composition – see text for discussion.

Figure 2 The goal structure at the beginning of the ADDTWO protocol where the subject makes up an example.

Figure 3 The goal structure for the portion of the protocol where the subject decides how to create a top-level function call that will be analogous to her desired program. The protocol starts with the goal, CODE EXAMPLE, and ends with successful mental simulation of CONS in order to check the code.

Figure 4 The goal structure governing the initial coding of ADDTWO. This starts at the goal, MAP TO ABSTRACT, and involves mapping the goal structure already built under CODE EXAMPLE to the definition.

Figure 5 The goal structure governing the testing and repair of the ADDTWO function. The structure under CHECK CODE is being generated to repair the code generated initially under MAP TO ABSTRACT.

Figure 6 The goal structure governing the episode in ADDTWO where the subject decides how to put an element into a list.

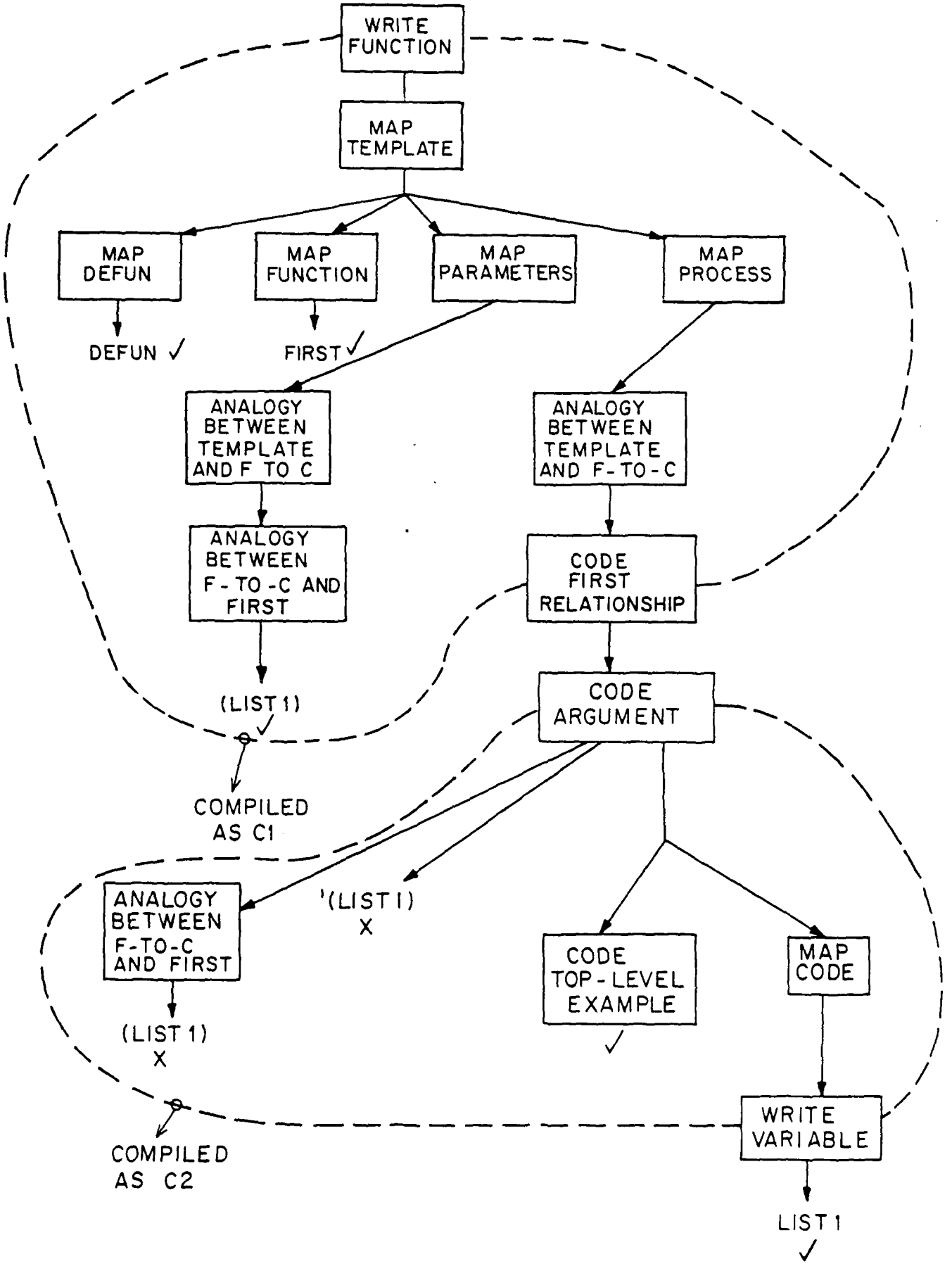
Figure 7 The goal structure governing the coding of ONETWO after the successful coding of ADDTWO.

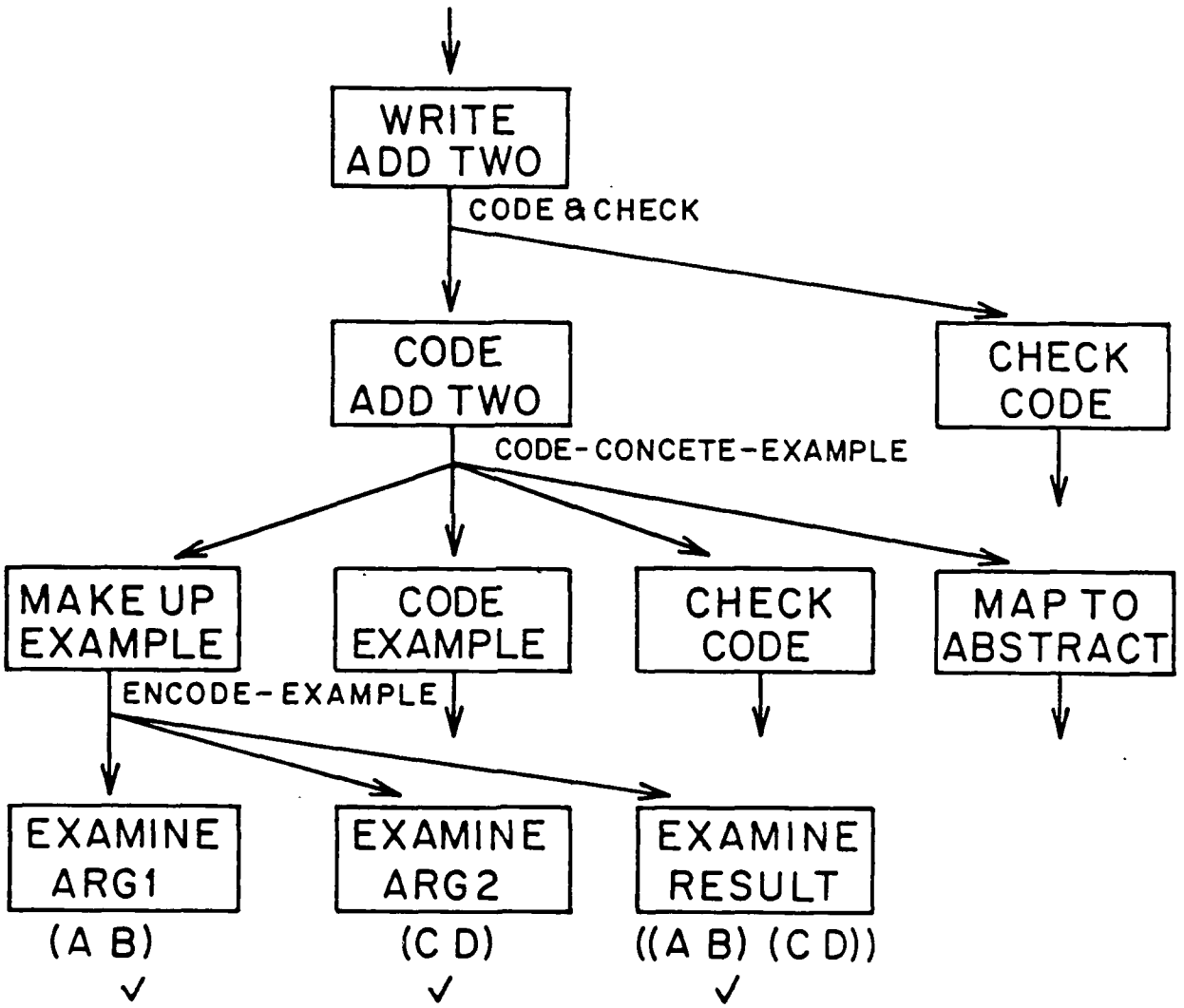
Figure 8 A specification of the POWERSET problem as presented to subjects.

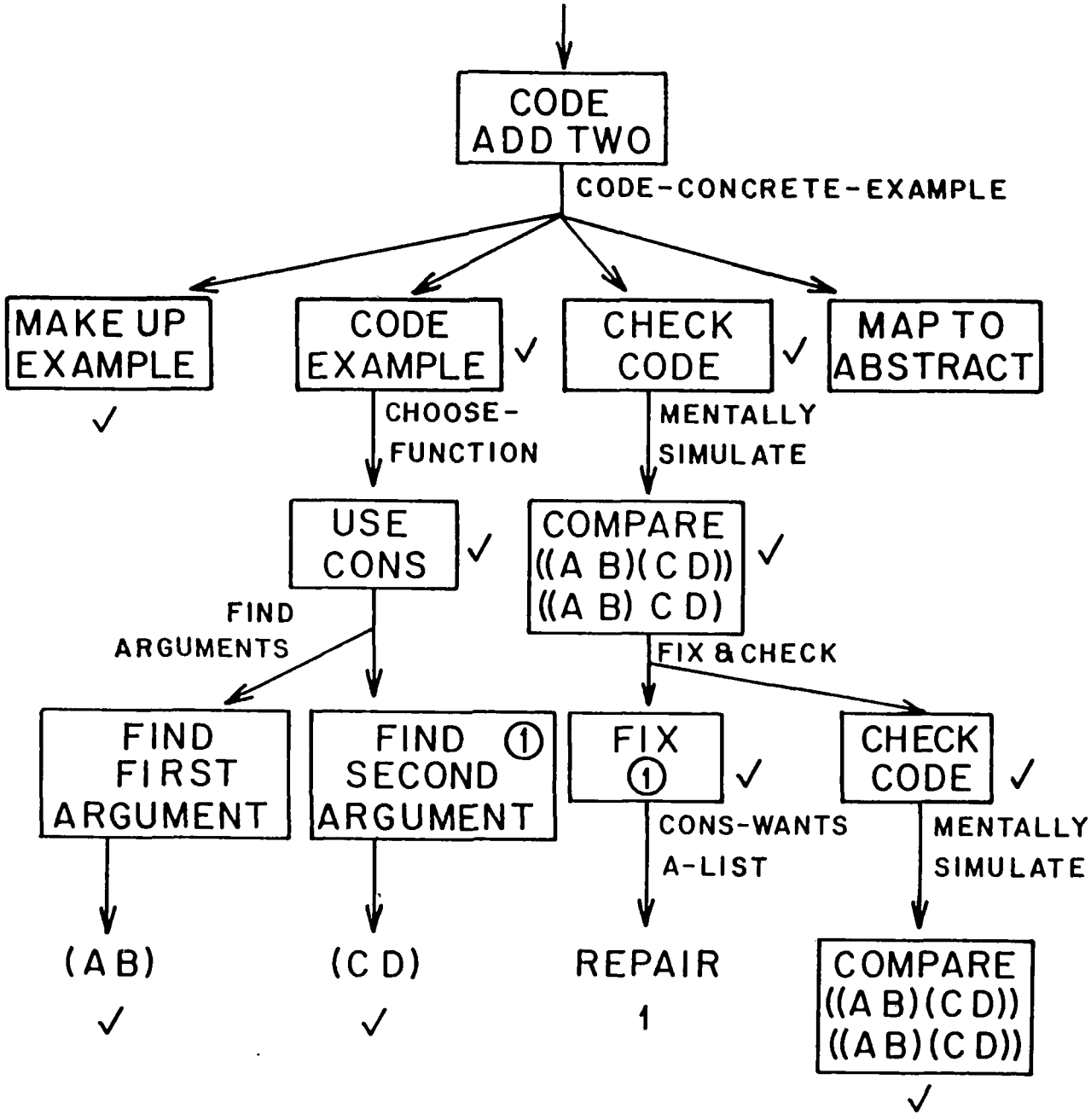
Figure 9 A representation of the essential insight which underlies solution of the POWERSET problem.

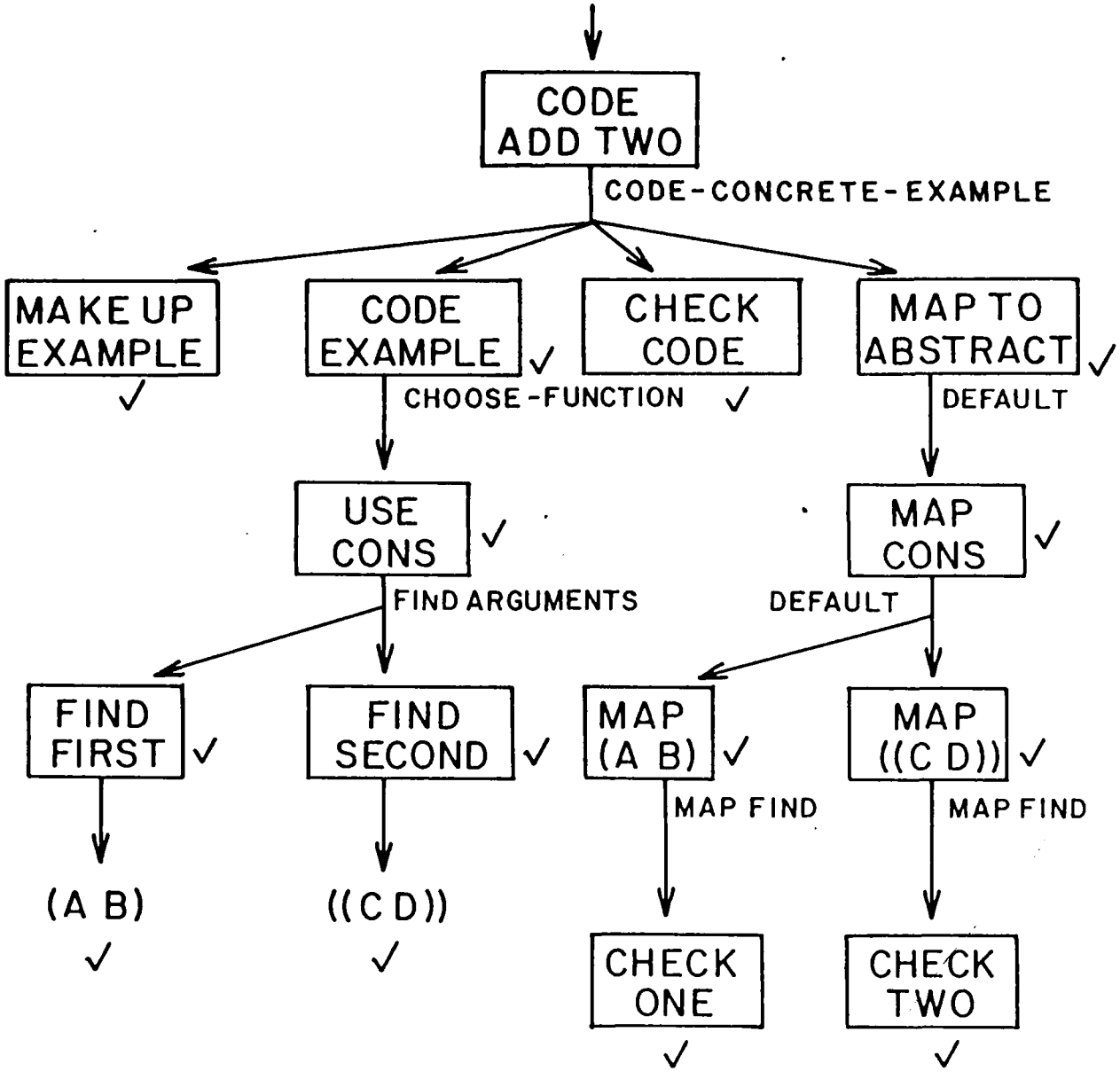
Figure 10 A representation of the hierarchical goal structure controlling GRAPES' solution of the POWERSET problem.

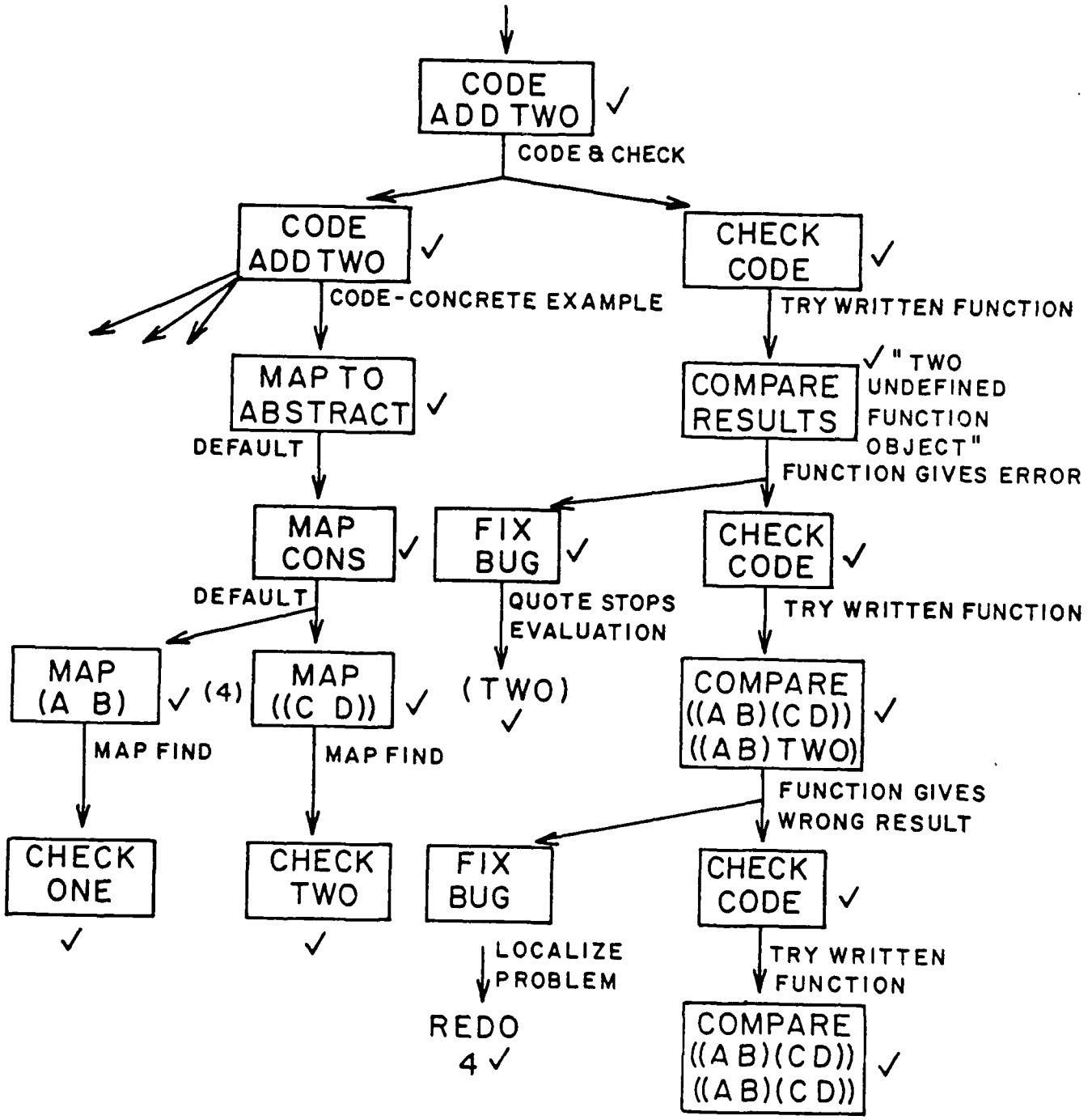
Figure 11 A representation of the hierarchical goal structure controlling GRAPES' solution of the ADDTO problem. This structure is a substructure of the goal structure in Figure 11.

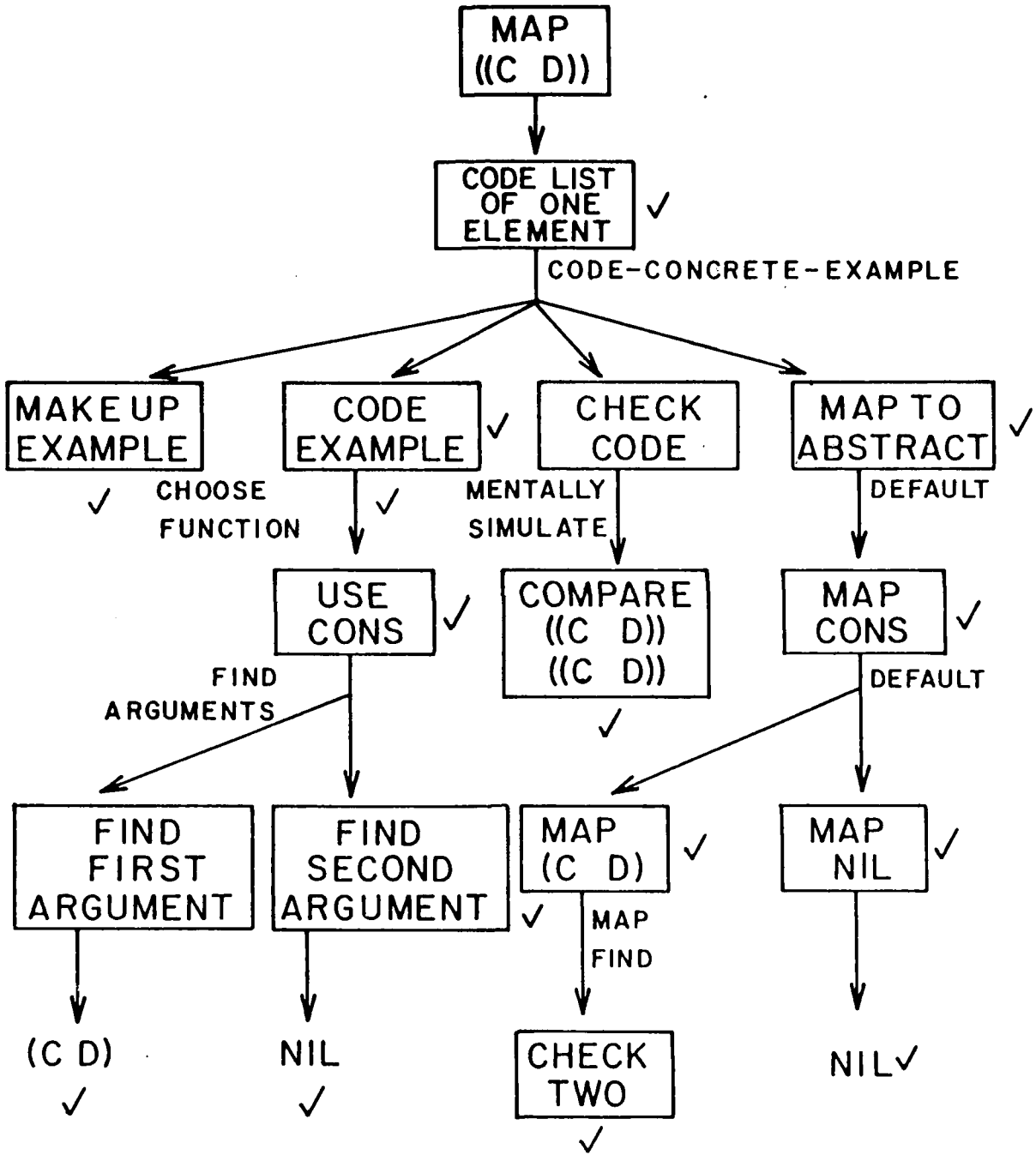

WRITE FUNCTION
MAP TEMPLATE
MAP DEFUN
MAP FUNCTION
MAP PARAMETERS
MAP PROCESS
DEFUN ✓
FIRST ✓
ANALOGY BETWEEN TEMPLATE AND F TO C
ANALOGY BETWEEN TEMPLATE AND F-TO-C
ANALOGY BETWEEN F-TO-C AND FIRST
CODE FIRST RELATIONSHIP
(LIST 1) ✓
CODE ARGUMENT
COMPILED AS C1
ANALOGY BETWEEN F-TO-C AND FIRST
'(LIST 1) X
CODE TOP-LEVEL EXAMPLE
MAP CODE
(LIST 1) X
WRITE VARIABLE
COMPILED AS C2
LIST 1 ✓

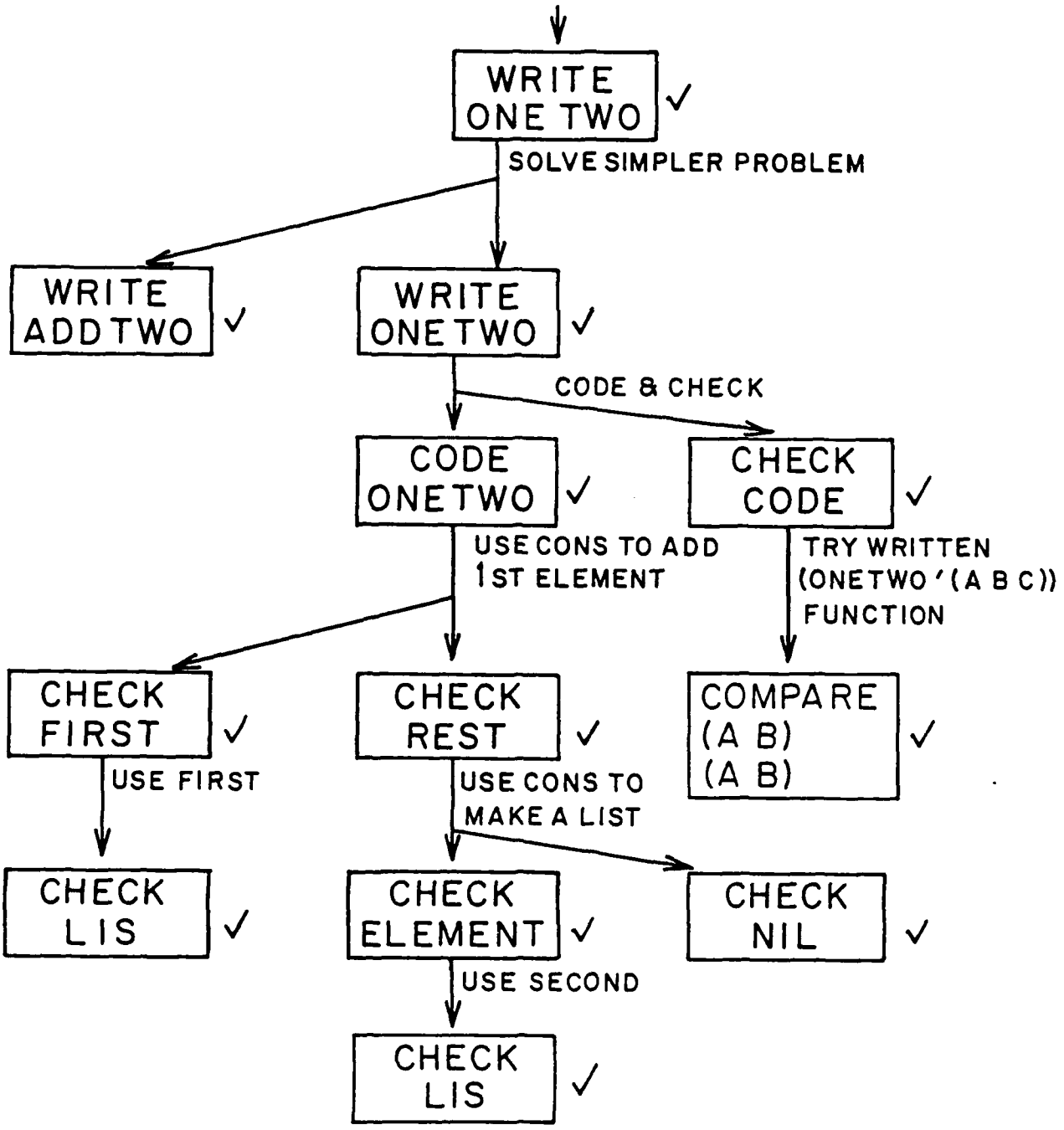

WRITE ADD TWO
CODE & CHECK
CODE ADD TWO
CHECK CODE
CODE-CONCETE-EXAMPLE
MAKE UP EXAMPLE
CODE EXAMPLE
CHECK CODE
MAP TO ABSTRACT
ENCODE-EXAMPLE
EXAMINE ARG1
EXAMINE ARG2
EXAMINE RESULT
(A B)
(C D)
((A B) (C D))


CODE ADD TWO
CODE-CONCRETE-EXAMPLE
MAKE UP EXAMPLE
✓
CODE EXAMPLE
✓
CHECK CODE
✓
MAP TO ABSTRACT
CHOOSE-FUNCTION
MENTALLY SIMULATE
USE CONS
✓
COMPARE ((A B)(C D)) ((A B) C D)
✓
FIND ARGUMENTS
FIX & CHECK
FIND FIRST ARGUMENT
FIND SECOND ARGUMENT ①
FIX ①
✓
CHECK CODE
✓
CONS-WANTS A-LIST
MENTALLY SIMULATE
(A B)
✓
(C D)
✓
REPAIR 1
COMPARE ((A B)(C D)) ((A B)(C D))
✓


CODE ADD TWO
CODE-CONCRETE-EXAMPLE
MAKE UP EXAMPLE
CODE EXAMPLE
CHECK CODE
MAP TO ABSTRACT
CHOOSE-FUNCTION
DEFAULT
USE CONS
MAP CONS
FIND ARGUMENTS
DEFAULT
FIND FIRST
FIND SECOND
MAP (A B)
MAP ((C D))
MAP FIND
MAP FIND
(A B)
((C D))
CHECK ONE
CHECK TWO


CODE ADD TWO
✓
CODE & CHECK
CODE ADD TWO
✓
CHECK CODE
✓
CODE-CONCRETE EXAMPLE
TRY WRITTEN FUNCTION
MAP TO ABSTRACT
✓
DEFAULT
COMPARE RESULTS
✓ " TWO UNDEFINED FUNCTION OBJECT"
FUNCTION GIVES ERROR
MAP CONS
✓
DEFAULT
FIX BUG
✓
QUOTE STOPS EVALUATION
CHECK CODE
✓
TRY WRITTEN FUNCTION
MAP (A B)
✓ (4)
MAP ((C D))
✓
(TWO)
✓
COMPARE ((A B)(C D)) ((A B) TWO)
✓
MAP FIND
MAP FIND
FUNCTION GIVES WRONG RESULT
CHECK ONE
✓
CHECK TWO
✓
FIX BUG
CHECK CODE
✓
LOCALIZE PROBLEM
TRY WRITTEN FUNCTION
REDO
4 ✓
COMPARE ((A B)(C D)) ((A B)(C D))
✓


MAP
((C D))
CODE LIST
OF ONE
ELEMENT
✓
CODE-CONCRETE-EXAMPLE
MAKE UP
EXAMPLE
✓
CHOOSE
FUNCTION
CODE
EXAMPLE
✓
✓
CHECK
CODE
MENTALLY
SIMULATE
MAP TO
ABSTRACT
✓
DEFAULT
USE
CONS
✓
FIND
ARGUMENTS
COMPARE
((C D))
((C D))
✓
MAP
CONS
✓
DEFAULT
FIND
FIRST
ARGUMENT
FIND
SECOND
ARGUMENT
MAP
(C D)
✓
✓
MAP
FIND
MAP
NIL
✓
(C D)
✓
NIL
✓
CHECK
TWO
✓
NIL✓


WRITE ONE TWO
✓
SOLVE SIMPLER PROBLEM
WRITE ADD TWO
✓
WRITE ONE TWO
✓
CODE & CHECK
CODE ONE TWO
✓
CHECK CODE
✓
USE CONS TO ADD 1ST ELEMENT
TRY WRITTEN (ONE TWO '(A B C)) FUNCTION
CHECK FIRST
✓
CHECK REST
✓
COMPARE (A B) (A B)
✓
USE FIRST
USE CONS TO MAKE A LIST
CHECK LIS
✓
CHECK ELEMENT
✓
CHECK NIL
✓
USE SECOND
CHECK LIS
✓

```
(POWERSET '(A B C)
  = (( A B C ) ( A B ) ( A C ) ( B C ) (A) (B) (C) ())
```

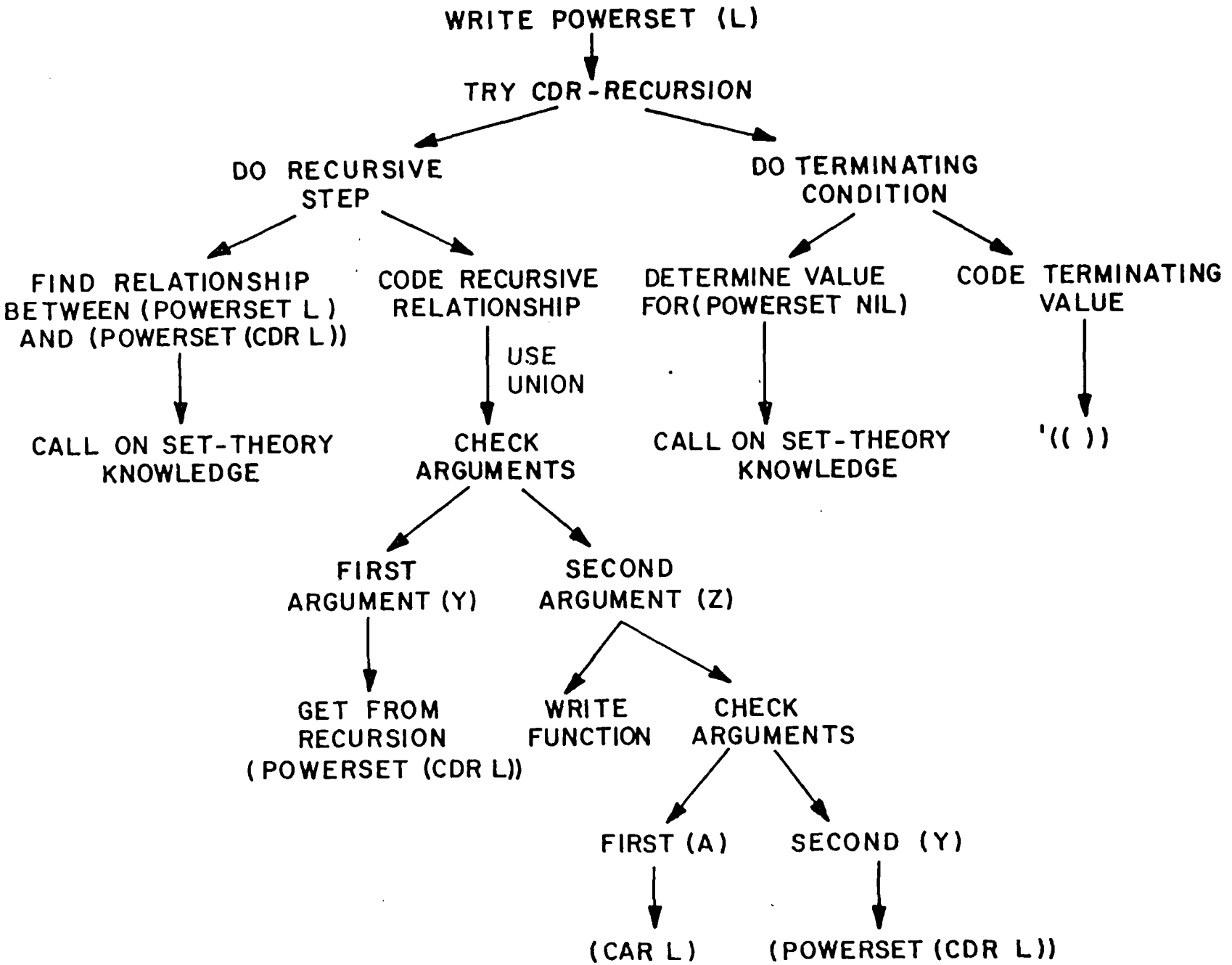
L= (A B C)

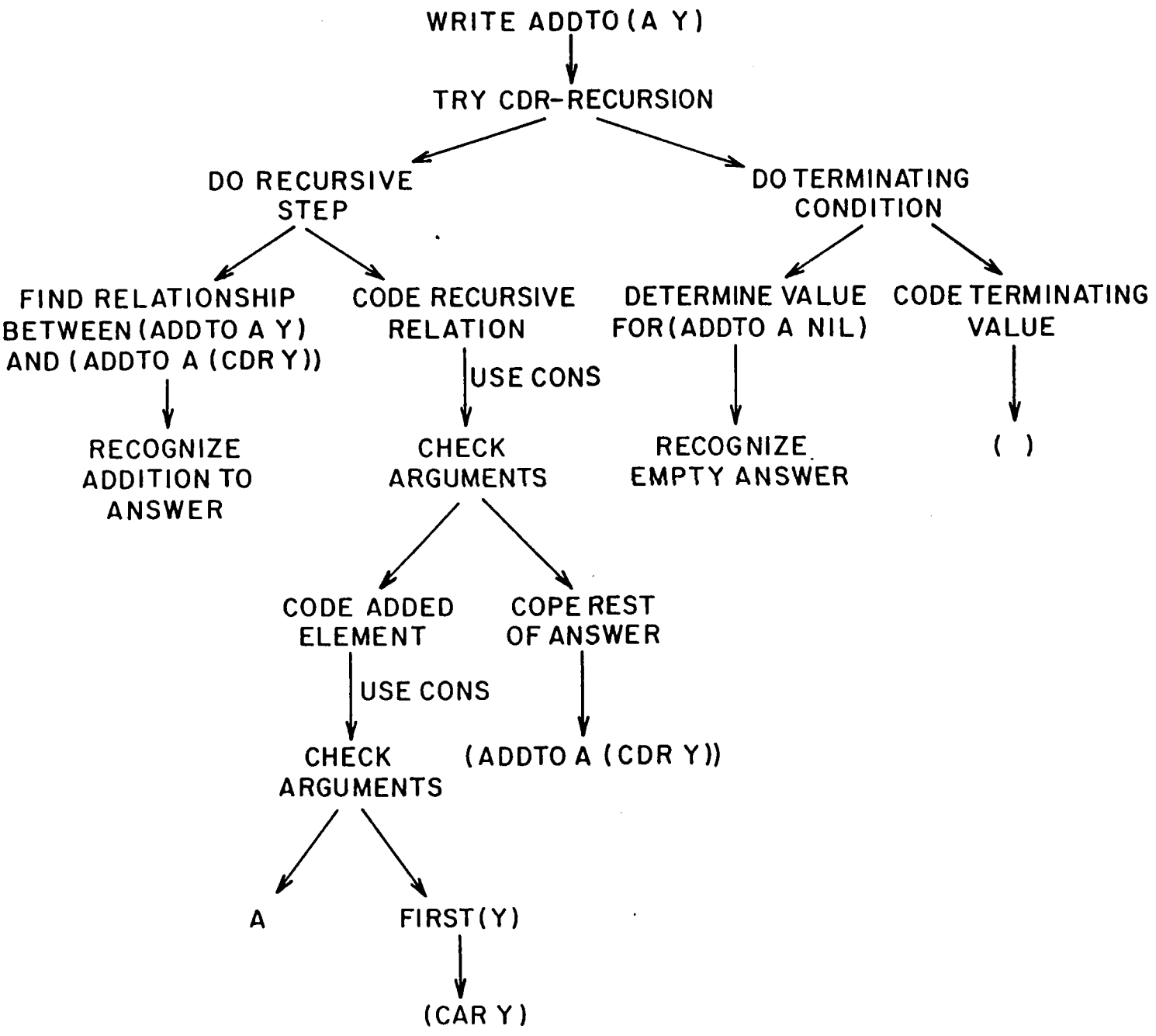
X= (POWERSET L)
= ((A B C)
(A B)
(A C)
(A)
(B C)
(B)
(C)
())

Y = (POWERSET (CDR L))
= ((B C)
(B)
(C)
())

X= Y+Z WHERE Z= ((A B C)
(A B)
(A C)
(A))

Z IS FORMED FROM Y BY ADDING A TO EACH MEMBER OF Y.


WRITE POWERSET (L)
TRY CDR-RECURSION
DO RECURSIVE STEP
DO TERMINATING CONDITION
FIND RELATIONSHIP BETWEEN (POWERSET L) AND (POWERSET (CDR L))
CODE RECURSIVE RELATIONSHIP
USE UNION
DETERMINE VALUE FOR(POWERSET NIL)
CODE TERMINATING VALUE
CALL ON SET-THEORY KNOWLEDGE
CHECK ARGUMENTS
CALL ON SET-THEORY KNOWLEDGE
'(( ))
FIRST ARGUMENT (Y)
SECOND ARGUMENT (Z)
GET FROM RECURSION (POWERSET (CDR L))
WRITE FUNCTION
CHECK ARGUMENTS
FIRST (A)
SECOND (Y)
(CAR L)
(POWERSET (CDR L))


WRITE ADDTO (A Y)
TRY CDR-RECURSION
DO RECURSIVE STEP
DO TERMINATING CONDITION
FIND RELATIONSHIP BETWEEN (ADDTO A Y) AND (ADDTO A (CDR Y))
CODE RECURSIVE RELATION
DETERMINE VALUE FOR (ADDTO A NIL)
CODE TERMINATING VALUE
USE CONS
RECOGNIZE ADDITION TO ANSWER
CHECK ARGUMENTS
RECOGNIZE EMPTY ANSWER
( )
CODE ADDED ELEMENT
COPE REST OF ANSWER
USE CONS
CHECK ARGUMENTS
(ADDTO A (CDR Y))
A
FIRST (Y)
(CAR Y)

Appendix A

R: This is the 6th session and this is February 8th. And we start by doing some problems. Do you want to review?

B: Yes, let me look through what we've done. Ok, we did setting functions, setting them up. Ok, you type DEFUN and then the function name, and then what the parameters will be and then the process. Ok, I'm just looking over the example for the temperature [?]...Then to exchange the lists. I think I know these. We'll see.

R: Ok, for the first one, why don't you read the problem, and then we'll do a slightly different--a subset of the first problem.

B: [B reads the problem 3-1 on page 37.]

R: Why don't we just define FIRST.

B: Ok, we start out by typing or writing DEFUN, and then the name, which would be FIRST. [long pause] I'm confused as to what the parameters would be.

R: You just have no ideas or you're choosing between several possibilities?

B: I don't know what to call--what--let me look at this again.

R: Ok, you're looking at?

B: At the temperature one. Ok, "when F-TO-C is used, it appears as the first element in a two element list. The second element is F-TO-C's argument. After the argument is evaluated, it becomes the temporary value of the function parameter. In this case, TEMP is the parameter, and it is given the value of the argument while F-TO-C is being evaluated." [From page 34.] Ok, that's no help. Ok, now I'm looking at the one about the pairs. [from problem 3-1] "Define new functions FIRST, REST, and INSERT that do the same things." Ok, in this they want [?]

R: But this is ours.

B: [from page 35] "This new function exchanges the first and second elements of a two-element list." Ok, so the function is called EXCHANGE. The parameter is PAIR, meaning two elements in the list. Ok, it makes a list out of those two [?]. Ok, DEFUN FIRST is what we want to call it, right?

R: Maybe I'll ask you, how many parameters do you think you'll need?

B: Just one, the first element of the list.

R: That's what the parameter is?

B: Or the list, I'm sorry.

R: Ok, the list. And what is the first element of the list--what relationship does that have to the function?

B: That's what you want to end up with.

R: Ok, that's called the value, the answer--what you end up with--in LISP is generally called the value.

B: Ok, so the parameter would be the list?

R: Yes.

B: Ok, so you type LIST [B has written (DEFUN FIRST (LIST)

R: Ok, well let me tell you something right now. That's a perfectly legitimate thing to do--to call a parameter 'LIST'--but because it has the same name as a function, it can make certain types of errors-- you'll never catch your error in a year because LISP will keep trying to make a list out of things, and so it's probably--LIS or LIS1 or something that's going to be a better name to call just to keep yourself from having con- fusion. But if you did everything right, it would be perfectly legitimate.

B: Maybe I could just call it LIST1.

R: Yes, that shouldn't cause anything, so L-I-S-T-1.

B: Ok.

R: Now where are you looking?

B: I was looking back over here.

R: Ok, the example INCREASE.

B: Ok, we want it to take the first element. We want it to do the same thing as CAR but it's going to be called something different. Can you use the function CAR? I'm not supposed to ask questions. Ok. Uh, ok, let me try this. Uh, let me look at this one again.

R: Ok, the EXCHANGE example.

B: Could it just be CAR and then LIST1?

R: We can try that.

B: Ok, we have the function itself, what it's going to be called, the parameter, what it's going to do. [B balances parentheses.]

R: Maybe we should try typing that in.

B: But I'm--there's nothing specific here so it wouldn't--I don't know...

R: I'm not--can you try and express it...

B: Ok, back here it just is the general definition of the function and it should return FIRST, I think. And then if we used a concrete example...

R: Why don't you right now give me an example--a concrete example we might type in after the function.

B: Ok, [B writes (FIRST (B R))] And it should return B.

[R types in the function definition: (DEFUN FIRST (LIST1) (CAR (LIST1))

B: Three closed parens.

R: Oh, sorry. [R adds a closed paren at the end.] And it says FIRST. Ok, now we're going to try FIRST of B R [(FIRST (B R))] ERROR: EVAL: undefined function B.

B: Ah, it should have a quote.

R: Ok, quote before the list. All right, FIRST QUOTE B R. [R types (FIRST '(B R)).] Ok, ERROR: EVAL: undefined function LIST1.

B: Maybe there should be a quote up here.

R: Ok, do you want to try that? That's now between the CAR and LIST1 is where you want to put the quote. Uh, maybe I'd like to ask you to tell me a little bit more about why you chose that.

B: Let me think about where the [?]. When I had just this without the quote, it saw B as just a function and this as its argument. [?] So then I put a quote here, and so that--and then it said that LIST1 wasn't recognized, right?

R: It said undefined function LIST1. LIST1 wasn't recognized as a function.

B: Uh, do you want to know why it doesn't?

R: Well, no, not necessarily. I want to know why you decided that quote would solve the problem.

B: Because a quote keeps whatever's after it from being looked at as a function.

R: All right, let's try seeing [?] quote. [R does some manipulations to make the change in the definition of the function to: (DEFUN FIRST (LIST1) (CAR '(LIST1))).] Ok, now we go back to FIRST of B R...Ok, it gave LIST1 as its answer.

B: It gave LIST1 as its answer. Hmm...interesting.

R: I presume that is not what you thought the answer was going to be.

B: Ok, up here I have CAR of the QUOTE and then some list. Uh...[long pause] [?] the list...I don't know what to do.

R: Ok, let me try [?] this way. Suppose we had typed in open paren CAR of QUOTE paren LIST1 closed paren closed paren [(CAR '(LIST1))], what would you expect to get as an answer to that?

B: Paren LIST1.

R: Should we try that? [R types it in.]

B: Or else just LIST1.

R: Ok, which one do you want to predict?

B: Uh, probably without the parentheses.

R: Ok, do you want to check it and [?]

B: Ok, I'm looking at page 23. Ok if you take the CAR of L you get A. Take the CAR of QUOTE L--QUOTE L means an L [?] I guess it would still be in parentheses--the list [?]

R: Ok, but what's the answer to this minor function [?]--what the CAR of QUOTE open paren LIST1

closed paren?

B: For this--I'm trying to think if it would have the parentheses or not. I don't think so.

R: Ok, so you think it's just LIST1 without the parentheses. Ok, does that give you any better understanding...

B: It has the parentheses or it doesn't?

R: It does not have parentheses.

B: So it actually returns LIST1. Ok, so here it returns LIST1. I don't--it's like it's not recognizing this at all.

R: It's not recognizing B. Let's try [?] that table here. Now this is going to have to be a little different because of F-TO-C--this is Figure 3-1, page 35--you're going to have FIRST, and instead of TEMP you're going to have--what's equivalent to what was TEMP in F-TO-C?

B: LIST1.

R: Ok, those are the only two things. Can we try doing the equivalent.

B: Ok, so you type in the function.

R: Ok, so you type in the function FIRST then B R.

B: Record current value of LIST1 if any. There isn't.

R: There is none. Ok.

B: So then you set LIST1 to B R. So LIST1 has the value B R. Use EVAL on body of F-TO--FIRST.

R: Ok, now what would happen if we did EVAL on open paren CAR of QUOTE open paren LIST1

closed paren closed paren? What would you get as the answer?

B: LIST1 without the parentheses.

R: Ok, all right. And that's exactly what happened. There were two more steps: Restore TEMP's value and Return value found to EVAL. Ok, does that help you understand why you got the wrong answer. It may not help you understand how you get the right answer, but does that help you understand why you got the wrong answer?

B: Yes, I think so. I still don't know how to get the right answer.

R: Let's assume that I had said to you 'Ok, we had just done SETQ LIST1 to QUOTE LIST B R. [(SETQ LIST1 '(B R))]. Now, how would you write the function to get the CAR of that, given that I just set LIST1 to...

B: You want me to do what?

R: To take the CAR.

B: Of this whole thing?

R: No, I want you to get the CAR of B R, given that it's been assigned to LIST1--that is, I don't want you to do it directly using B R, I want you to do it using LIST1.

B: Ok, so you've--you've already--you've set LIST1 equal to B R. Then it would be CAR...are we doing this like--assuming this is what would be up here? You want to know how I would get...

R: No, we're just doing it totally independently. I'm just saying [?] Just like with an exercise before, remember we had to get PEAR out of [?] Get B out of LIST1. Yes, it's equivalent to that one.

B: Ok, so...at this point, LIST1 has the value B R.

R: You wrote down CAR...

B: CAR LIST1, I guess. Wait. I'm looking back [?]

R: Ok, you wrote CAR open paren LIST1 closed paren closed paren.

B: Ok, here we set L--we used quotes there, we didn't use QUOTE here.

R: That's because we did SETQ.

B: We set L to the list A B, and we've set the LIST1 to the list A B--the list B R. Ok, then we did the CAR of L there and they got A out. So I think that might work.

R: Do you see a difference between what you've written and what they've written?

B: Yes. I have parentheses.

R: Ok, do you think that will make a difference?

B: Yes, it would probably just bring back LIST1. So I would want it to be CAR LIST1. [(CAR LIST1)]

R: Ok, let's try it. [R types in the SETQ function and the CAR function.] And you get B back. So does that...now...

B: So this is the correct one to get B?

R: Ok, from--assuming that the list B R had been set to LIST1. Now does that give you any ideas of how you might want to modify your definition of FIRST?

B: Ok, I was looking at this while you were typing.

R: Ok, you were looking at [?] F-TO-C.

B: F-TO-C, where they SETQ TEMP and they did something else with it.

R: You think that's going to help?

B: Maybe not.

R: If it is, that's what I want to know--why?

B: No, maybe not.

R: I didn't mean to imply [?]

B: No, actually what they're doing I guess is setting the temperature to the difference between the temperature and 32, and that's not quite what we want to do. Ok, this was the way-- after we had set LIST1 to B R, and then we did CAR of LIST1, we got B. If we had had just CAR and LIST1 up here...and then just typed in FIRST and parentheses B R--I'm not sure if I need to use SETQ in the definition.

R: Ok, do you remember where the idea of doing the SETQ came from?

B: What do you mean? Where in the book?

R: In the book in the sense of we were talking about this figure, 3-1.

B: Oh, ok. So that's sort of inherent in the function.

R: Yes. [?]

B: Ok, so I don't have to do that. Ok, so let's try this. [B writes: (DEFUN FIRST (LIST1) (CAR LIST1)), and R types it in to the computer.]

R: And now you want me to do FIRST of QUOTE B R.

B: Well, I didn't put a quote in.

R: Oh, you don't want to put a quote in?

B: I'm not sure, since you put it in.

R: I was just copying it from here. Which would you like to do? Ok, so you want to do it--do you want to try it without the quote?

B: Without the quote, it would probably do the same thing as before.

R: Which was?

B: It would look at this as a function.

R: Oh, B [?]

B: So it should have a quote.

R: Ok, and it comes back with B. Ok, do you want to try it on anything else?

B: No. I mean--I assume it's the same--any kind of list would do that.

R: Ok, let's do as a second thing, let's try to define a function called SECOND that--is it clear what SECOND would do?

B: Yes. Ok, DEFUN SECOND, the parameter would be LIST1--can I still call it that?

R: Yes.

B: Ok, I'm just going to make up a list--assume you have the list X Y and Z-- And you want to get the Y out.

R: What are those funny marks between the letters?

B: Commas. They're not supposed to be there, right?

R: I just couldn't figure out what they were. Not in general.

B: Ok, and you want to get Y, so you'd want to take the CDR of this to end up with Y and Z, and then take the CAR and end up with Y. So, you'd have CDR of the LIST1, and out here you'd have your CAR. Ok, I want to look back to where we did multiple CDRs and CARS. Let's see...ok, I'm looking on page 20. Ok, CAR CDR and then QUOTE and then a list. B is what's returned. [at the bottom of page 21.] Ok, that's what I want it to do. [B starts reading the next-to-the-last paragraph on page 21.] Ok, I want it to do CDR. So I think...I think I just want it like that. [the function is: (DEFUN SECOND (LIST1) (CAR (CDR LIST1)))] I'll write an example.

[R has problems getting the computer to work properly.]

R: So, it looks like SECOND is okay. I won't bother to explain to you how I know.

[1]This research is supported by contract N00014-81-C-0335 from the Office of Naval Research. We would like to thank Robin Jeffries both for many hours of valuable discussions relevant to the research and for her comments on the paper. We would also like to thank Lynne Reder and Gordon Bower for their valuable comments on the manuscript.

[2]All programs in LISP take the form of functions that calculate a particular input-out relation.

[3]Here and throughout the paper we will give English-like rendition of the production rules. A technical specification of these rules (i.e., a computer listing) can be obtained by writing to us. Also available is a users' manual (Sauers & Farrell, 1982) that describes the system.

[4]The LISP function CDR returns all of a list except its first member, i.e. (CDR '(A B C)) = (B C).

[5]In LISP, functions can be directly typed into the monitor and directly applied or they can be part of function definitions, in which case they are applied only when the function is evaluated.

[6]Many have commented that they felt it was unintuitive to claim that procedures could be created from a single problem-solving episode. Their intuition is that it should take much longer to proceduralize knowledge. Rather, they suspect that the subject is using various declarative traces to guide the solution of the second problem. There is no hard evidence in the protocols on this matter. However, the one-trial learning position is consistent with a frequent report students' give in solving a series of problems (e.g., in calculus text). This is that they find the second problem much easier and they have no idea why. This lack of introspective awareness of the cause of the improvement is to be predicted from the procedural position.

[7]This essentially involved breaking in on the simulation and changing working memory – i.e., we have no real theory of why this working memory error occurred, only what its consequences were.

[8]Given the more advanced stage of SS, the function syntax, parenthesis balancing, etc. are all parts of compiled productions – unlike the previous simulation of BR.

[9]Note that she has made the same error as BR in placing an argument in parentheses but for very different reasons. SS follows the error message with the same repair as used by BR. Her comment at the point of the error message is "Must have something to do with when we defined that and I put parentheses around *two*. Cause there's no *two* in there, let's see...Maybe I should have...what would it do if I quoted that?"

[10]Because of GRAPES specificity conflict resolution principle, more specific rules, like MAP-FIND, will apply before less specific, default rules like the rule applied here.

[11]And in fact, Peter Pirolli at Carnegie-Mellon University has written a GRAPES simulation of how this insight is uncovered

[12]Unfortunately, this asymmetry is confounded with the fact that "LIST" is more mnemonic as a function name than CONS. However, we have also done an experiment that reversed the function names and still found the effect—i.e., now "CONS" is better than "LIST" (See Anderson, 1983b).

[13]Anderson (1982) argues for a process of tuning in addition to knowledge compilation. This is a mechanism by which operators, once compiled, become more appropriate in their range of application. For instance, there are many places where CDR-recursion could be tried. On only some of these would it be appropriate. We see very little of this tuning in our protocols, perhaps because we are only looking at the first 30 hours of learning.

Navy

1 Robert Ahlers
Code N711
Human Factors Laboratory
NAVTRAEQUIPCEN
Orlando, FL 32813

1 Dr. Meryl S. Baker
Navy Personnel R&D Center
San Diego, CA 92152

1 Code N711
Attn: Arthur S. Blaiwes
Naval Training Equipment Center
Orlando, FL 32813

1 Liaison Scientist
Office of Naval Research
Branch Office, London
Box 39
FPO New York, NY 09510

1 Dr. Richard Cantone
Navy Research Laboratory
Code 7510
Washington, DC 20375

1 Dr. Stanley Collyer
Office of Naval Technology
800 N. Quincy Street
Arlington, VA 22217

1 CDR Mike Curran
Office of Naval Research
800 N. Quincy St.
Code 270
Arlington, VA 22217

1 Dr. Tom Duffy
Navy Personnel R&D Center
San Diego, CA 92152

1 DR. PAT FEDERICO
Code P13
NPRDC
San Diego, CA 92152

1 Dr. Jude Franklin
Code 7510
Navy Research Laboratory
Washington, DC 20375

Navy

1 Dr. Mike Gaynor
Navy Research Laboratory
Code 7510
Washington, DC 20375

1 LT Steven D. Harris, MSC, USN
RFD 1, Box 243
Riner, VA 24149

1 Dr. Jim Hollan
Code 14
Navy Personnel R & D Center
San Diego, CA 92152

1 Dr. Ed Hutchins
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Norman J. Kerr
Chief of Naval Technical Training
Naval Air Station Memphis (75)
Millington, TN 38054

1 Dr. Peter Kincaid
Training Analysis & Evaluation Group
Dept. of the Navy
Orlando, FL 32813

1 Dr. James Lester
ONR Detachment
495 Summer Street
Boston, MA 02210

1 Dr. William L. Maloy (02)
Chief of Naval Education and Training
Naval Air Station
Pensacola, FL 32508

1 Dr. Joe McLachlan
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr William Montague
NPRDC Code 13
San Diego, CA 92152

1 Technical Director
Navy Personnel R&D Center
San Diego, CA 92152

6 Commanding Officer
Naval Research Laboratory
Code 2627
Washington, DC 20390

Navy

1 Office of Naval Research
Code 433
800 N. Quincy SStreet
Arlington, VA 22217

6 Personnel & Training Research Group
Code 442PT
Office of Naval Research
Arlington, VA 22217

1 Office of the Chief of Naval Operations
Research Development & Studies Branch
OP 115
Washington, DC 20350

1 LT Frank C. Petho, MSC, USN (Ph.D)
CNET (N-432)
NAS
Pensacola, FL 32508

1 Dr. Gil Ricard
Code N711
NTEC
Orlando, FL 32813

1 Dr. Robert G. Smith
Office of Chief of Naval Operations
OP-987H
Washington, DC 20350

1 Dr. Alfred F. Smode, Director
Training Analysis & Evaluation Group
Dept. of the Navy
Orlando, FL 32813

1 Dr. Richard Sorensen
Navy Personnel R&D Center
San Diego, CA 92152

1 Dr. Frederick Steinheiser
CNO - OP115
Navy Annex
Arlington, VA 20370

1 Roger Weissinger-Baylon
Department of Administrative Sciences
Naval Postgraduate School
Monterey, CA 93940

1 Mr John H. Wolfe
Navy Personnel R&D Center
San Diego, CA 92152

Navy

1 Dr. Wallace Wulfeck, III
Navy Personnel R&D Center
San Diego, CA 92152

Marine Corps

1 H. William Greenup
Education Advisor (E031)
Education Center, MCDEC
Quantico, VA 22134

1 Special Assistant for Marine
Corps Matters
Code 100M
Office of Naval Research
800 N. Quincy St.
Arlington, VA 22217

1 DR. A.L. SLAFKOSKY
SCIENTIFIC ADVISOR (CODE RD-1)
HQ, U.S. MARINE CORPS
WASHINGTON, DC 20380

Army

1 Technical Director
U. S. Army Research Institute for the
Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Mr. James Baker
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Milton S. Katz
Training Technical Area
U.S. Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Marshall Narva
US Army Research Institute for the
Behavioral & Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Harold F. O'Neil, Jr.
Director, Training Research Lab
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Commander, U.S. Army Research Institute
for the Behavioral & Social Sciences
ATTN: PERI-BR (Dr. Judith Orasanu)
5001 Eisenhower Avenue
Alexandria, VA 20333

1 Joseph Psotka, Ph.D.
ATTN: PERI-1C
Army Research Institute
5001 Eisenhower Ave.
Alexandria, VA 22333

1 Dr. Robert Sasmor
U. S. Army Research Institute for the
Behavioral and Social Sciences
5001 Eisenhower Avenue
Alexandria, VA 22333

1 Dr. Robert Wisher
Army Research Institute
5001 Eisenhower Avenue
Alexandria, VA 22333

Air Force

1 Technical Documents Center
Air Force Human Resources Laboratory
WPAFB, OH 45433

1 U.S. Air Force Office of Scientific Research
Life Sciences Directorate, NL
Bolling Air Force Base
Washington, DC 20332

1 Air University Library
AUL/LSE 76/443
Maxwell AFB, AL 36112

1 Dr. Earl A. Alluisi
HQ, AFHRL (AFSC)
Brooks AFB, TX 78235

1 Mr. Raymond E. Christal
AFHRL/MOE
Brooks AFB, TX 78235

1 Bryan Dallman
AFHRL/LRT
Lowry AFB, CO 80230

1 Dr. Alfred R. Fregly
AFOSR/NL
Bolling AFB, DC 20332

1 Dr. Genevieve Haddad
Program Manager
Life Sciences Directorate
AFOSR
Bolling AFB, DC 20332

1 Dr. T. M. Longridge
AFHRL/OTE
Williams AFB, AZ 85224

1 Dr. John Tangney
AFOSR/NL
Bolling AFB, DC 20332

1 Dr. Joseph Yasatuke
AFHRL/LRT
Lowry AFB, CO 80230

Department of Defense

12 Defense Technical Information Center
Cameron Station, Bldg 5
Alexandria, VA 22314
Attn: TC

1 Military Assistant for Training and Personnel Technology
Office of the Under Secretary of Defens for Research & Engineering
Room 3D129, The Pentagon
Washington, DC 20301

1 Major Jack Thorpe
DARPA
1400 Wilson Blvd.
Arlington, VA 22209

Civilian Agencies

1 Dr. Patricia A. Butler
NIE-BRN Bldg, Stop # 7
1200 19th St., NW
Washington, DC 20208

1 Dr. Susan Chipman
Learning and Development
National Institute of Education
1200 19th Street NW
Washington, DC 20208

1 Edward Esty
Department of Education, OERI
MS 40
1200 19th St., NW
Washington, DC 20208

1 Dr. John Mays
National Institute of Education
1200 19th Street NW
Washington, DC 20208

1 Dr. Arthur Melmed
724 Brown
U. S. Dept. of Education
Washington, DC 20208

1 Dr. Andrew R. Molnar
Office of Scientific and Engineering
Personnel and Education
National Science Foundation
Washington, DC 20550

1 Chief, Psychological Reserch Branch
U. S. Coast Guard (G-P-1/2/TP42)
Washington, DC 20593

1 Dr. Frank Withrow
U. S. Office of Education
400 Maryland Ave. SW
Washington, DC 20202

1 Dr. Joseph L. Young, Director
Memory & Cognitive Processes
National Science Foundation
Washington, DC 20550

Private Sector

1 Dr. Patricia Baggett
Department of Psychology
University of Colorado
Boulder, CO 80309

1 Mr. Avron Barr
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Dr. John Black
Yale University
Box 11A, Yale Station
New Haven, CT 06520

1 Dr. John S. Brown
XEROX Palo Alto Research Center
3333 Coyote Road
Palo Alto, CA 94304

1 Dr. Glenn Bryan
6208 Poe Road
Bethesda, MD 20817

1 Dr. Bruce Buchanan
Department of Computer Science
Stanford University
Stanford, CA 94305

1 Dr. Jaime Carbonell
Carnegie-Mellon University
Department of Psychology
Pittsburgh, PA 15213

1 Dr. Pat Carpenter
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. William Chase
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Micheline Chi
Learning R & D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. William Clancey
Department of Computer Science
Stanford University
Stanford, CA 94306

Private Sector

1 Dr. Michael Cole
University of California
at San Diego
Laboratory of Comparative
Human Cognition - D003A
La Jolla, CA 92093

1 Dr. Allan M. Collins
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. Lynn A. Cooper
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15213

1 Dr. Paul Feltovich
Department of Medical Education
Southern Illinois University
School of Medicine
P.O. Box 3926
Springfield, IL 62708

1 Professor Reuven Feuerstein
HWCRI Rehov Karmon 6
Bet Hakerem
Jerusalem
Israel

1 Mr. Wallace Feurzeig
Department of Educational Technology
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02238

1 Dr. Dexter Fletcher
WICAT Research Institute
1875 S. State St.
Orem, UT 22333

1 Dr. John R. Frederiksen
Bolt Beranek & Newman
50 Moulton Street
Cambridge, MA 02138

1 Dr. Michael Genesereth
Department of Computer Science
Stanford University
Stanford, CA 94305

Private Sector

1 Dr. Don Gentner
Center for Human Information Processing
University of California, San Diego
La Jolla, CA 92093

1 Dr. Dedre Gentner
Bolt Beranek & Newman
10 Moulton St.
Cambridge, MA 02138

1 Dr. Robert Glaser
Learning Research & Development Center
University of Pittsburgh
3939 O'Hara Street
PITTSBURGH, PA 15260

1 Dr. Josph Goguen
SRI International
333 Ravenswood Avenue
Menlo Park, CA 94025

1 Dr. Daniel Gopher
Department of Psychology
University of Illinois
Champaign, IL 61820

1 Dr. Bert Green
Johns Hopkins University
Department of Psychology
Charles & 34th Street
Baltimore, MD 21218

1 DR. JAMES G. GREENO
LRDC
UNIVERSITY OF PITTSBURGH
3939 O'HARA STREET
PITTSBURGH, PA 15213

1 Dr. Barbara Hayes-Roth
Department of Computer Science
Stanford University
Stanford, CA 95305

1 Dr. Frederick Hayes-Roth
Teknowledge
525 University Ave.
Palo Alto, CA 94301

1 Dr. Earl Hunt
Dept. of Psychology
University of Washington
Seattle, WA 98105

Private Sector

1 Dr. Marcel Just
Department of Psychology
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. Scott Kelso
Haskins Laboratories, Inc
270 Crown Street
New Haven, CT 06510

1 Dr. David Kieras
Department of Psychology
University of Arizona
Tuscon, AZ 85721

1 Dr. Walter Kintsch
Department of Psychology
University of Colorado
Boulder, CO 80302

1 Dr. Stephen Kosslyn
1236 William James Hall
33 Kirkland St.
Cambridge, MA 02138

1 Dr. Pat Langley
The Robotics Institute
Carnegie-Mellon University
Pittsburgh, PA 15213

1 Dr. Jill Larkin
Department of Psychology
Carnegie Mellon University
Pittsburgh, PA 15213

1 Dr. Alan Lesgold
Learning R&D Center
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 15260

1 Dr. Jim Levin
University of California
at San Diego
Laboratory fof Comparative
Human Cognition - D003A
La Jolla, CA 92093

1 Dr. Michael Levine
Department of Educational Psychology
210 Education Bldg.
University of Illinois
Champaign, IL 61801

Private Sector

1 Dr. Marcia C. Linn
Lawrence Hall of Science
University of California
Berkeley, CA 94720

1 Dr. Jay McClelland
Department of Psychology
MIT
Cambridge, MA 02139

1 Dr. James R. Miller
Computer*Thought Corporation
1721 West Plano Highway
Plano, TX 75075

1 Dr. Mark Miller
Computer*Thought Corporation
1721 West Plano Parkway
Plano, TX 75075

1 Dr. Tom Moran
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

1 Dr. Allen Munro
Behavioral Technology Laboratories
1845 Elena Ave., Fourth Floor
Redondo Beach, CA 90277

1 Dr. Donald A Norman
Cognitive Science, C-015
Univ. of California, San Diego
La Jolla, CA 92093

1 Dr. Jesse Orlansky
Institute for Defense Analyses
1801 N. Beauregard St.
Alexandria, VA 22311

1 Prof. Seymour Papert
20C-109
Massachusetts Institute of Technology
Cambridge, MA 02139

1 Dr. Nancy Pennington
University of Chicago
Graduate School of Business
1101 E. 58th St.
Chicago, IL 60637

Private Sector

1 DR. PETER POLSON
DEPT. OF PSYCHOLOGY
UNIVERSITY OF COLORADO
BOULDER, CO 80309

1 Dr. Fred Reif
Physics Department
University of California
Berkeley, CA 94720

1 Dr. Lauren Resnick
LRDC
University of Pittsburgh
3939 O'Hara Street
Pittsburgh, PA 1521

1 Mary S. Riley
Program in Cognitive Science
Center for Human Information Processing
University of California, San Diego
La Jolla, CA 92093

1 Dr. Andrew M. Rose
American Institutes for Research
1055 Thomas Jefferson St. NW
Washington, DC 20007

1 Dr. Ernst Z. Rothkopf
Bell Laboratories
Murray Hill, NJ 07974

1 Dr. William B. Rouse
Georgia Institute of Technology
School of Industrial & Systems
Engineering
Atlanta, GA 30332

1 Dr. David Rumelhart
Center for Human Information Processing
Univ. of California, San Diego
La Jolla, CA 92093

1 Dr. Michael J. Samet
Perceptronics, Inc
6271 Variel Avenue
Woodland Hills, CA 91364

1 Dr. Roger Schank
Yale University
Department of Computer Science
P.O. Box 2158
New Haven, CT 06520

Private Sector

1 Dr. Walter Schneider
Psychology Department
603 E. Daniel
Champaign, IL 61820

1 Dr. Alan Schoenfeld
Mathematics and Education
The University of Rochester
Rochester, NY 14627

1 Mr. Colin Sheppard
Applied Psychology Unit
Admiralty Marine Technology Est.
Teddington, Middlesex
United Kingdom

1 Dr. H. Wallace Sinaiko
Program Director
Manpower Research and Advisory Services
Smithsonian Institution
801 North Pitt Street
Alexandria, VA 22314

1 Dr. Edward E. Smith
Bolt Beranek & Newman, Inc.
50 Moulton Street
Cambridge, MA 02138

1 Dr. Richard Snow
School of Education
Stanford University
Stanford, CA 94305

1 Dr. Eliott Soloway
Yale University
Department of Computer Science
P.O. Box 2158
New Haven, CT 06520

1 Dr. Kathryn T. Spoehr
Psychology Department
Brown University
Providence, RI 02912

1 Dr. Robert Sternberg
Dept. of Psychology
Yale University
Box 11A, Yale Station
New Haven, CT 06520

1 Dr. Albert Stevens
Bolt Beranek & Newman, Inc.
10 Moulton St.
Cambridge, MA 02238

Private Sector

1 David E. Stone, Ph.D.
Hazeltine Corporation
7680 Old Springhouse Road
McLean, VA 22102

1 DR. PATRICK SUPPES
INSTITUTE FOR MATHEMATICAL STUDIES IN
THE SOCIAL SCIENCES
STANFORD UNIVERSITY
STANFORD, CA 94305

1 Dr. Kikumi Tatsuoka
Computer Based Education Research Lab
252 Engineering Research Laboratory
Urbana, IL 61801

1 Dr. Maurice Tatsuoka
220 Education Bldg
1310 S. Sixth St.
Champaign, IL 61820

1 Dr. Perry W. Thorndyke
Perceptronics, Inc.
545 Middlefield Road, Suite 140
Menlo Park, CA 94025

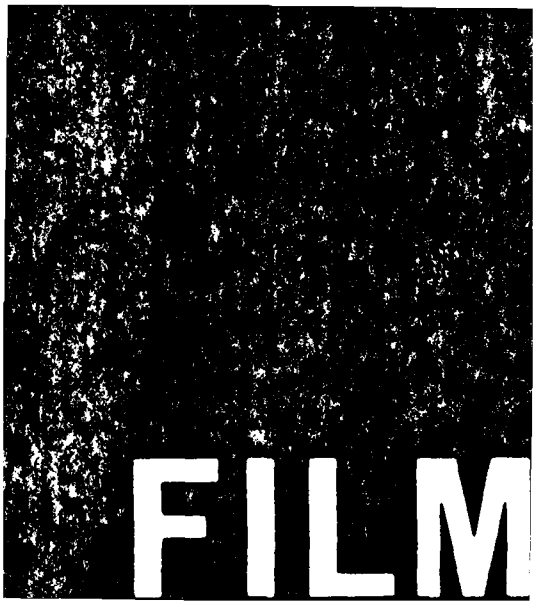
1 Dr. Douglas Towne
Univ. of So. California
Behavioral Technology Labs
1845 S. Elena Ave.
Redondo Beach, CA 90277

1 Dr. Kurt Van Lehn
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304

1 Dr. Keith T. Wescourt
Perceptronics, Inc.
545 Middlefield Road, Suite 140
Menlo Park, CA 94025

1 Dr. Thomas Wickens
Perceptronics, Inc.
6271 Variel Ave.
Woodland Hills, CA 91364

1 Dr. Mike Williams
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304


FILM